Design and Implementation of the Parallel Functional Language Eden

Dissertation
zur
Erlangung des Doktorgrades
der Naturwissenschaften
(Dr. rer. nat.)

dem
Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
vorgelegt von

Silvia Breitinger
aus München

Marburg/Lahn 1998

Vom Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg als Dissertation am 15. Juli 1998 angenommen.

Erstgutachterin: Prof. Dr. Rita Loogen
Zweitgutachter: Prof. Dr. Herbert Kuchen, Universität Münster
Tag der mündlichen Prüfung am 23. Juli 1998

Contents

Zusammenfassung

In der vorliegenden Arbeit wird die explizit parallele funktionale Sprache **Eden** definiert. Die Sprache wird als Koordinierungsmodell auf die bedarfsgesteuerte funktionale Standardsprache Haskell aufgesetzt. Dieser Ansatz weist gegenüber bisher existierenden Ansätzen zur parallelen Programmierung entscheidende Vorteile auf: Zum einen wird die Programmentwicklung und -analyse dadurch erleichtert, zum anderen kann bei der Implementierung auf bereits bestehende Komponenten zurückgegriffen werden.

Die wesentlichen Merkmale des Koordinationsmodells sind die explizite Handhabung von Prozessen, die implizite Behandlung von Kommunikation und die gezielte Einführung von spekulativer Parallelität. Zu diesem Zweck werden sogenannte Prozeßabstraktionen und Prozeßinstantiierungen definiert, die die Entsprechung der Abstraktionen und Applikationen im λ-Kalkül bilden. Diese Elemente bilden die funktionale Kernsprache, in der die referentielle Transparenz erhalten bleibt.

Um die Ausdrucksmächtigkeit der Sprache speziell für sogenannte reaktive Systeme zu erweitern, werden darüber hinaus weitere Sprachelemente eingeführt. Diese erlauben insbesondere das Arbeiten mit Zeitabhängigkeiten.

Eine strikte Trennung zwischen den beiden Sprachebenen erleichtert die Analyse von Programmen mit Hilfe formaler Methoden. Beispielsweise kann durch Untersuchung syntaktischer Merkmale auf möglichen Nichtdeterminismus in Programmen geschlossen werden. Die operationale Semantik der Sprache arbeitet explizit mit den beiden Ebenen und führt über sogenannte Aktionen eine wohldefinierte Schnittstelle zwischen ihnen ein. Die Syntax und Semantik der Sprache wird in Teil I der Arbeit detailliert behandelt.

Der zweite Teil der Arbeit beschreibt die Implementierung von Eden. Nach einer Diskussion der Anforderungen an eine Implementierung und einer Klassifikation der zugrundegelegten Abstraktionsebenen werden zwei Ansätze zur Implementierung im Detail beschrieben: einerseits eine Prototypversion, die mit Skeletten für statische Prozeßsysteme arbeitet und andererseits eine parallele abstrakte Maschine und deren Implementierung. Laufzeitmessungen mit dem ersten Ansatz haben gezeigt, daß die explizite Handhabung von Prozeßsystemen für die Effizienz paralleler Algorithmen vorteilhaft ist.

Der abschließende dritte Teil diskutiert Anwendungen und dokumentiert auf diese Weise die vielfältigen Einsatzmöglichkeiten der Sprache.

Chapter 1

Introduction

1.1 Parallelism

Parallel computers are used in order to gain more speed than can be obtained with uniprocessors. This is mandatory for a number of application areas. Nowadays the prices of microprocessors are decreasing steadily [Sla96] and parallel computers are widely available. But the promised advent of parallel computing has been fulfilled only partly. This can be attributed to two factors: on the one hand, the development of suitable software is lagging behind, and on the other hand, the development of specific hardware is extremely costly.

Although a variety of different designs for parallel computers have been proposed in the last twenty years, most large-scale parallel computers available today are distributed memory machines[Sch98]. The advantage of this design is that standard processors can be used, but such machines are hard to program because of their high communication latencies. Communication between the processing elements can either be realized by message passing, or by defining a layer of shared information by software [TKB92].

There exists a large number of approaches to parallel programming, which in different ways use a sequential language as their basis and extend it with constructs for parallelism. In general, making too many aspects of parallelism explicit can render programming complicated and error-prone. From the user's point of view, it is of course desirable to have parallelizing systems or compilers where the parallelism is transparent.

Some approaches to this goal are already in use today. For instance, the instruction-level parallelism present in RISC processors is not visible to the programmer. There is a continuing migration from transparent low-level parallelism to higher-level parallelism, i.e. from bit parallelism [CS97] to instruction level parallelism [HP96, LW92] and thread parallelism [LH94].

Nevertheless it cannot be expected that such a transparent system with even higher-level, i.e. *process-level*, parallelism can be developed in the next ten years. Parallelizing compilers exist, but are restricted to simple and local forms of parallelism like loop- or pipeline parallelism [ZC91].

In summary, inherent parallelism can be exploited in certain areas, but the speedup produced by it is not powerful enough at the moment. In particular, parallel programming and the design of dedicated parallel algorithms is not superseded by these approaches.
In general, designing an efficient and correct parallel program is much more complicated than designing an equivalent sequential one. Most parallel programs in use today are

based on low-level models like MPI (Message passing interface [MPI94]) communication routines.

Several high level approaches to simplify parallel programming have been proposed: e.g. High Performance Fortran, data parallel programming languages[Brä89, Ble96a], the bulk-synchronous programming (BSP) model[Val90], and algorithmic skeletons[Col89, DFH+93].

But all these models impose restrictions on the kind of parallelism supported or on the kind of programs that can be expressed, or both. A number of languages have been successful in a special application area. They are however very restricted, so that they cannot be used for parallel software development in a general context. They aim at a specialized application area instead, such as scientific computing [Ske91, Sch97a].

1.2 Functional programming

The frequently stated benefits of functional languages are their side-effect freedom, their polymorphic Hindley-Milner type system and their high level of abstraction [BW88].

Although the functional approach dates back to the 1960s, it has been restricted to academic use for a long period of time. This could be attributed to a number of shortcomings of existing functional languages and their implementations which inhibited their use for large-scale applications. By now, the state of the art has advanced considerably, so that functional languages can be used for practically all areas of application:

1. Standard languages have been established[PH97].
2. There exist efficient implementations, which approach the speed of C [HFA+96].
3. There exist attractive approaches for the handling of side-effects and I/O [Wad97].
4. There exist extensions which provide the programmer with added flexibility, such as (multi-parameter) type classes[PJM97], libraries and unboxed values[PL91].
5. Space requirements can be limited: Update in place can be handled. Much progress has been made in the area of garbage collection[Moh97].
6. Interfaces to imperative languages are available[NP96].

Implicit parallelism is a form of parallelism inherently present in a language, such as the parallelism resulting from independent subexpressions in functional languages or the and / or - parallelism in logic languages. Implicitly parallel approaches can also provide annotations for controlling the parallelism. For these languages, the semantics of the annotated program is identical to the semantics of the program without annotations.

Implicitly parallel functional approaches have been an active field of research[Ham94]. It has however turned out that parallel speedups are hard to achieve because the (useful) parallelism is hard to detect, so that granularity problems are incurred. This particularly applies to their implementation on distributed memory computers.

Explicit parallelism however is found in languages with special constructs for which a *sequential* semantics is not defined, i.e. concurrency is reflected in the semantics.

We distinguish between two classes of concurrent systems[Sha89]: *transformational systems* are closed in the sense that they receive some initial input data and generate a

final result. *Reactive systems*, by contrast, are not mainly intended to compute a result, but to interact with their environment. Due to the necessity for time-dependent reactions, these systems have to be non-deterministic and in general also non-terminating. Based on this classification, parallel systems are defined as a special case of *concurrent* systems: concurrent systems with transformational behaviour are called *parallel*.

Concurrency can elegantly be expressed by introducing two distinct language levels: The lower level, which is also called *computation language*, is formed by a sequential programming language. The upper level, the so-called *coordination language*, introduces and controls concurrency. The isolation of this language level has the advantage that non-functional constructs can be adopted without affecting the semantics of the sequential part.

This *separation* approach was advocated in a more general context by Gelernter and Carriero [GC92], who establish Linda as an all-purpose coordination language that can be combined with various computation languages. They identify the use of such an *orthogonal* coordination language as a prerequisite for a clear and general representation of communication issues.

1.3 The approach

The above discussion has shown that it is one of the central issues in parallel programming today to strike a balance between explicit and implicit mechanisms. The problem to be solved in this thesis is to provide a framework for the development of parallel algorithms endowed with the following properties:

1. High level of abstraction
2. Generality and flexibility
3. Support for formal methods
4. Support for efficient implementation
5. Modelling of both transformational and concurrent behaviour.

This dissertation presents the parallel functional language Eden[BLOP96a]. It reports on the syntax, semantics, implementation and pragmatics of the language. The language defines a novel approach to high-level parallel programming. In a more general context, it addresses the question "how explicit should parallel functional programming be?" The language Eden is presented, which uses explicit definitions for processes and topologies, but not for communication operations and process placement (see also [BLOP96b, BLOP97a, BLO96, BLO95, BLOP97b]).

Alternatively, a tool is presented which comprises additional explicit components, but is still based on functional programming (cf. [BLP98]).

Eden overcomes the shortcomings of implicitly parallel functional approaches by providing more clarity about parallel processes and the interconnections between them. In this way, Eden alleviates the flow of information from the programmer to the implementation, while at the same time retaining the high level of abstraction of functional languages.

This forms a useful framework in order to strike a balance between programming efficiency and runtime efficiency. Rapid prototyping and stepwise refinement are supported in this way. Preliminary runtime measurements demonstrate not only the viability of the approach, but also the benefits of the design in comparison to implicitly parallel languages (cf. [BL98, BKL+98b, BKL97b, BKL98a]).

Structure of this dissertation

This thesis consists of three parts, which describe the syntax and semantics (Part I), the implementation (Part II) and possible applications of the language (Part III):

Part I forms the central part of the thesis by defining the language Eden and its operational semantics, together with a detailed discussion of its expressibility. Part II explains the implementation of Eden. Part III discusses the application of Eden in two different areas, namely numerical algorithms and simulation programming. They do not present whole application projects, but analyze Eden's potential to handle these areas, and thereby provide an outlook on the use of Eden for programming in the large.

Acknowledgements Many people have provided valuable support and in this way made this thesis possible. First of all, I would like to thank my supervisor Rita Loogen, who gave much encouragement and valuable advice on many topics. Thanks also go to my colleagues Yolanda Ortega Mallén, Ricardo Peña, Ulrike Klusik, Herbert Kuchen, Steffen Priebe, Hendrik Lock and Manuel Chakravarty for discussions related to the language Eden. In addition, the support by the DAAD German-Spanish *acción integrada* is gratefully acknowledged.

Last but not least I want to thank my fiancé Olaf Gutberlet and my parents and grandparents for their completely non-scientific kind of support.

Part I

Syntax and Semantics of the language Eden

Chapter 2

Design of the Language Eden

Basic principles of Eden. Eden has been designed to have the best of two worlds: on the one hand, the high level of abstraction and side effect freedom of a lazy functional language and on the other hand the clear view of process systems present in languages with concurrency or explicit parallelism. The two key features of Eden's design are its explicit notion of a process and its controlled treatment of time-dependencies. In this way, a new language is defined that maintains the declarative reading of functional programs while enhancing its expressibility considerably. The extensions are defined in a way that allows a clean semantical treatment of the language.

Eden extends the lazy functional language Haskell[PH97] with a coordination language. In the following sections, we will explain the central features of this coordination language in detail.

2.1 Processes

An Eden process comprises a set of functions[1] that map input data to output data. It can have several input- and output-channels. Following λ-abstractions and applications, we define process abstractions and process instantiations.

2.1.1 Process abstractions

A process abstraction defines a general process scheme with parameters.

Syntax. A process abstraction for the mapping of inputs $in_1, \ldots, in_m$ to outputs $out_1, \ldots, out_n$ is specified by:

$$\boxed{\begin{array}{lcl} p\ par_1 \ldots par_k & = & \texttt{process}\ (in_1, \ldots, in_m)\ \texttt{->}\ (out_1, \ldots, out_n) \\ & & \texttt{where}\ equation_1 \ldots equation_r \end{array}}$$

The tuples of inputs and outputs after the keyword `process` provide information about the *interface* of the process. The symbol `->` indicates the mapping of inputs to outputs. For $m = 1$ and $n = 1$, the tuple notation can be omitted. The definitions following the `where` form the *body* of the process abstraction. Together, they define the computations

[1] Not a function with multiple results

performed by the process in a functional way. The roles of parameters and inputs will be explained in Chapter 2.1.2.

Type. Eden extends the polymorphic type system of Haskell by the predefined binary type constructor *Process* which is used to type process abstractions. The above process abstraction has the following type:

$$\boxed{p :: \tilde{\tau}_1 \rightarrow \ldots \rightarrow \tilde{\tau}_k \rightarrow \mathit{Process}\ (\tau_1, \ldots, \tau_m)\ (\tau'_1, \ldots, \tau'_n),}$$

where $\tilde{\tau}_1, \ldots, \tilde{\tau}_k$ denote the types of the parameters and $\tau_1, \ldots, \tau_k$ and $\tau'_1, \ldots, \tau'_m$ are the types of the inputs and of the outputs, respectively.

Process abstractions as first-class citizens. A process abstraction can also have functions or process abstractions as inputs or outputs. Process abstractions are first-class citizens, i.e. they can be passed as function arguments and can be produced as function results. The only exception to this is a restriction explained in Chapter 2.5.

2.1.2 Process instantiation

The creation of Eden processes is explicit. Process creation can be carried out when a process abstraction with no (more) unbound parameters is *instantiated*, i.e. applied to a tuple of input expressions.

Syntax. The infix operator `#` takes a parameterless process abstraction and a tuple of input expressions and creates a new process. The resulting Eden expression is called *process instantiation*:

$$\boxed{(\,p\ e_1 \ldots e_k\,)\ \texttt{\#}\ (\,\mathit{input\_exp}_1, \ldots, \mathit{input\_exp}_m\,)}$$

Type. The instantiation operator has type `(#) :: Process a b -> a -> b`. As the result of the instantiation is the tuple of outputs generated, the type of the process instantiation is simply the type of these outputs. Consequently there is no particular type that represents a *running* process.

The input and output types can be marked with *channel annotations* that specify the mode of transmission (see Chapter 2.2.3), but the presence of such annotations does not change the *type* of the information transferred. Such annotations do neither influence the body of the process nor the type inference.

Parameters and inputs. When applied to a sufficient number of parameters and a tuple of input expressions, a process abstraction yields a tuple of outputs. If there are multiple outputs, the corresponding evaluations are performed by separate threads running concurrently in the same process.

The tuple which wraps up the inputs of a process indicates that the corresponding expressions must be provided "in one piece" while currying is possible for the process parameters.

Often, process instantiations occur in the following form of equations defining the tuple of outputs of the newly created process:

$$(\,o_1, \ldots, o_n\,) = (\,p\ e_1 \ldots e_k\,)\ \texttt{\#}\ (\,\mathit{input\_exp}_1, \ldots, \mathit{input\_exp}_m\,)$$

The left hand side of such an equation must be a tuple of terms which matches the outputs of the created process.

On process creation, communication channels between the parent process and the newly generated child are built up. The parent supplies the child with inputs and receives its outputs. Communication in a process system will be discussed in detail below.

2.1.3 Process systems

Eden does neither allow the transmission nor the duplication of running processes. As we presuppose a lazy semantics, it is guaranteed that *process outputs* used as arguments in other expressions will be shared and not computed more than once.

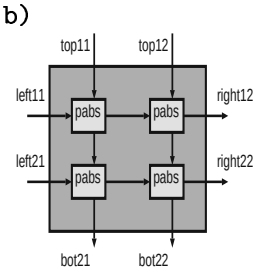
We have explained before that every Eden process can instantiate child processes dynamically. Moreover, processes can always communicate with their ancestors and descendants. Arbitrary communication topologies can be defined (see also Chapter 3.2.3). In particular, we allow the immediate creation of topologies by supporting the creation of several child processes at the same time (see Chapter 2.3). The names of output channels denote the edges in the corresponding process graph.

Even complicated topologies can be created very conveniently by instantiating a suitable abstraction ("skeleton") for this topology. The subsequent example shows the definition of such a frequently used skeleton. A further concept that alleviates the construction of complex communication topologies is the dynamic reply channel mechanism explained in Chapter 2.4.

Example 2.1

(2×2 grid of processes). The process abstraction `grid2` generates a grid of 4 processes. The process abstraction for the cell processes is passed as parameter `pabs`. The channels are oriented from left to right and from top to bottom.

```
grid2 :: Process (a,b) (a,b) -> Process (a,a,b,b) (a,a,b,b)
grid2 pabs
  = process (left11,  left21,  top11, top12) ->
    (right12, right22, bot21, bot22)
    where (right11, bot11) = pabs # (left11,  top11)
          (right12, bot12) = pabs # (right11, top12)
          (right21, bot21) = pabs # (left21,  bot11)
          (right22, bot22) = pabs # (right21, bot12)
```



◁

2.2 Communication

2.2.1 Basic principles

As Eden is tailored for distributed memory systems, it uses message passing as underlying principle for communication. In order to support the hiding of communication latencies, asynchronous message passing is chosen.

Communication channels. Processes use unidirectional 1:1 communication channels, which are distinguished by unique names. In most cases, these channels are automatically established on process creation: Parent and child process are provided with in- and outputs for the exchange of input data and results.

Implicit communication. In Eden, no explicit send and receive operations have to be inserted into the code. The interface of a process lists the names of inputs and outputs. The sending of data is then simply triggered by an equation with the name of an output on the left hand side, i.e. the *definition* of this output. Likewise, the reading of input is encoded by equations that contain the names of inputs on the right hand side, i.e. the *access* to this input. Prior to its transmission, data is always evaluated to *normal form* by the sender process.

Asynchronous message passing. As the underlying communication paradigm is the asynchronous exchange of messages, we work with nonblocking send and blocking receive. A receive operation blocks on empty input, i.e. a thread that tried to access it is suspended until input arrives.
Generally, communication in Eden is assumed to be reliable.

2.2.2 Simple communication channels

There are two types of communication channels, *streams* and *one-value* channels. There are two important principles for communication which are common to these channels:

1. Arbitrary data types can be transmitted.

2. Data is evaluated by the sender and transmitted in normal form.

One value channels have an Eden data type (a so-called *message type*, see Chapter 4.2.4) as their type and they transmit one expression of this type. Their observable behaviour is that this expression is communicated using *one message*.
Stream channels transmit lists in a way that especially supports interleaving of computations in the producer and consumer. The observable behaviour of a stream is the *one-by-one transmission* of the elements.

The sender evaluates the list to be output to normal form and pieces of this list are transmitted as soon as possible. In this way, streams form head-strict lazy lists. In particular, the programmer can use streams in order to implement systems with mutual recursion over lazy lists.

Application and implementation aspects of these two kinds of channels will be discussed in Chapters 3.2.2 and 6.1.1. In the following we will explain which type of channel is used in which case and present a generalization of communication channels.

2.2.3 Channel structures

In addition to the simple communication channels presented above, Eden also supports data structures with channels as their components.

Defaults for channels. For programming convenience, in- and outputs of type list are by default interpreted as *streams*. If the output is not of type list, a *one value* channel of the corresponding type is used.

Channel annotations. In cases where the above default interpretation is not desired, the system can be told to build up different connections by placing the mixfix annotations '<' '>' around the types that should be interpreted as individual channels[2]. For instance, the annotated type `[<a>]` represents a list of channels of type `a`.

These *channel annotations* have to be specified in the *type declaration* of the process abstraction (see Chapter 2.1.1), which in the default case is optional.

As these annotations do not change the type of the data, `a` and `<a>` indicate the same type. Inside the body of a process, it is irrelevant how many channels are used for transmission, e.g. if a list of stream channels (`[<[a]>]`) or one stream of lists (`[[a]]`) is used, the result being of type `[[a]]` in both cases. But in the presence of infinite or undefined substructures, this distinction is of importance from an operational point of view.

Obviously, simply convening that the outermost list structure were always to be transmitted in the form of a stream would have been much too restrictive. One of the (not very frequent) cases where we have to guarantee that *several* channels are used instead of one, will be the `merge` process explained in Chapter 2.5.

2.3 Coordinating laziness and parallelism

In Eden, we combine lazy evaluation with a form of strictness in the coordination constructs. In this way, parallel computations become faster and easier to handle than entirely demand-driven parallel computations, without having to sacrifice the advantages of lazy functional languages. In the following, we will explain the details of this convention.

Creation of processes. We call a process instantiation *top-level* if it occurs directly in equations of the form *var* = *instantiation* in the body of a process abstraction, without any surrounding expressions. Such process abstractions are created "spontaneously"[3]. In contrast to this, process instantiations embedded in other expressions do not receive any special treatment, i.e. they are only executed on demand.

For process instantiations which are not on the top level, it is evident that they have to produce output, since otherwise there could not be demand leading to their creation. In general, every process has to have a statically introduced output. But this output can also be trivial, i.e. `()`.

Parameters and the distribution of work. Parameters differ from inputs in the time and method of transmission. Firstly, process parameters are passed before process creation, while input values are passed after process creation. Secondly, process parameter expressions are copied into the body of the process abstraction and are on demand evaluated by the created process itself. This is important because for an efficient implementation on systems with distributed memory, processes have to be *closed* objects

[2] A simple normalization step that extracts this interface information will be described in Chapter 4.3.1. Due to the above conventions, it is not possible to specify that one list *has to* be transmitted in one piece, which would anyway be useless from an implementation point of view.

[3] To be precise, all process instantiations on the top level of the *Normalized Kernel Eden* version of a program are executed spontaneously. This set contains all process instantiations already on the top level in the original program. The programmer has to keep in mind that *all input expressions* of a process are going to be evaluated, even if the child process does not access this data at all.

without any implicit sharing of information. Input channel values are evaluated by the sender and received across the communication channel.

Output. In Eden, by default there is demand for the outputs of a process. This means, a process expects its outputs to be useful unless it is told otherwise. This could be the case if the consumer of the output is either too slow to consume it at the moment or it is no longer existent. These cases can efficiently be handled, see below.

Termination. If a process has fully evaluated some output, or the receiver of the data will no longer consume it, the corresponding outport and thread will be deactivated. A *process* terminates if there are no outports left, because then it has nothing to do any more.

2.4 Dynamic reply channels

The previous sections presented the functional core of Eden. In the next two sections, we will present extensions which help to handle time-dependencies.

In the following we will present a construct that especially supports the handling of partial information. In addition to the dynamic creation of processes, a process may generate a new *input channel* and send a message containing the name of this channel to another process.

A process may use a received channel name either to return some information to the sender process, or pass the channel name further on to another process. These possibilities exclude each other and are termed *use once* and *pass once*, respectively. The programmer should ensure that each dynamically created channel is used to establish a one-to-one connection between a unique writer process and the generator of the channel. This restriction cannot be checked statically. A runtime error would be raised if a single reply channel were used more than once[4].

In this way, arbitrary processes can exchange references to channels via their existing communication channels and then create new connections for communicating directly. Subsequently we will describe the syntactic constructs introduced for this.

2.4.1 Creation of reply channels

We introduce a new unary type constructor `Chan_name` for the names of dynamically created reply channels. A new channel name of type `Chan_name` τ and a corresponding input channel of type τ are declared in the expression

`new (`*ch_name*`, `*chan*`) exp`

The scope of the newly generated channel *chan* and its name *ch_name* is the body expression `exp`. The object *ch_name* is a reference to the newly created input channel *chan*. This means, if *ch_name* is transferred to a prospective sender process, *chan* will receive its reply.

[4] This restriction does of course not preclude stream communication.

2.4.2 Connection to reply channels

A process receiving a reply channel name *ch_name* and wanting to reply through it, uses an expression of the following form:

$$\boxed{\textit{ch\_name}\ \texttt{!*}\ \textit{expression}_1\ \texttt{par}\ \textit{expression}_2}$$

Before $expression_2$ is evaluated, a new concurrent thread for the evaluation of $expression_1$ is generated. The result of this concurrent evaluation is sent via the reply channel given as first argument. The result of the whole expression is $expression_2$. The connection between sender and receiver will be established when the above expression is evaluated. At that time it will be checked that this reply channel is not already connected.

Example 2.2

In this example generic master and worker process abstractions are presented. The master uses a function `collect` to both gather the results generated by the workers and distribute the remaining work to the workers that finished their previous work. The messages sent by the workers always contain the result generated and a reply channel for requesting more work. The nullary constructor `ZeroWork` has type `Work` and is used to encode the fact that no work is left.

```
master :: Process ([Work], [(Result, Chan_name Work)]) [Result]
master =  process (items, inp) -> collect items inp
          where
          collect :: [a] -> [(b, Chan_name a)] -> [b]
          collect []        [] =    []  -- no items left, no requests
          collect []        ((result, next):inRest)
                = next !* ZeroWork   par  (result : collect []    inRest)
          collect (i:items) ((result, next):inRest)
                = next !* i          par  (result : collect items inRest)
```

The workers receive messages from the master that contain an item of new work. This item is composed with the worker's own item `bs` by means of a function `comp`, which both are given as parameters. The worker receives one item as initial work and then always receives a further work item when returning the result to the previous one.

```
worker :: (Work -> b -> Result) -> b ->
          Process (Work) [(Result, Chan_name (Work))]
worker     comp myB
  = process as -> work as
    where work ZeroWork = []            -- no work, empty result
          work item     = new(chan, task) (comp item myB, chan) : work task
```

◁

The above simple example has shown a situation where the use of dynamic channels is not mandatory, but convenient. Moreover, it exemplifies the typical use of this construct.

Synchronization of outputs. In general, sending of data is implicit: as soon as the normal form of the result is available, the data is sent. In the presence of multiple outputs, data dependencies have to ensure that output is carried out at the intended point of time (see Chapter 3.3.2). If there are special synchronization requirements, these can be expressed in a natural way with the above `par`-expression. Note that the connection to the channel forms a side effect, cf. Chapter 3.3.3 for a discussion.

2.5 Fair many-to-one communication

One of the goals of Eden is the modelling of real-world concurrent applications with time dependencies. In this application area, the following requirements have to be met.

Time-dependent reaction. Consider the situation of the `master` process in the previous example. If this process is connected to multiple worker processes, it is vital that it is able to react to incoming messages in a time dependent manner.

With the constructs defined so far, the master process would have to have the outputs of the workers as separate inputs. But "simultaneous" waiting for input arriving across multiple input channels can not be modelled. In a purely functional specification, one would always have to select one input channel that is inspected next. If no input were available on this particular input, the whole computation would be bound to block. It is not possible to *test* for incoming input. But even if a non-blocking test for incoming input could be performed, a second problem would have to be solved:

Fairness. The second requirement arising in the presence of multiple inputs is *fairness*. If there are several channels that contain much input, care must be taken that none of them is ignored indefinitely. A makeshift solution to this problem would be the use of alternating tags which indicate the input to be attended to next. For this mechanism we would again need a *test* or time-out mechanism in order to switch the tag in the following cases: Firstly, if the selected input were empty and others weren't, the unnecessary blocking would have to be prevented. Secondly, if the input channel tested first contained an infinite amount of input, the starvation of the others would have to be prevented.

This discussion shows that combinations of *test* or time-out constructs and alternating selection flags would generate solutions which were neither elegant nor declarative. For this reason, in Eden a different amendment is chosen to remedy the shortcomings of the purely functional approach: the introduction of a predefined merge agent which meets both the above requirements.

A non-functional process abstraction

With the introduction of a predefined process abstraction `merge`, we enhance the language's expressibility by a feature for time-dependency. This extension is carried out in a very controlled way, so that referential transparency inside of functions is not lost.

This process abstraction can be used to create a nondeterministic *fair* merging process for a list of stream channels with type `[msg]` each.

```
merge  ::  Process  [<[msg]>]  [msg]
```

This `merge` process abstraction implements fair many-to-one communication[5]. If a process is supposed to communicate with several other processes in a fair way without fixing a deterministic order in which processes are attended, a `merge` process can be used to transform the list of outputs from the requesting processes into a single input stream.

[5]Note that here the channel annotations ensure that the inputs are transmitted using separate channels, which is vital in order to receive the inputs in a fair way.

Example 2.3

With `merge`, we can now complete the master–worker example with a system that creates the master process and the worker processes. It employs `merge` in order to transform the outputs of the workers into one input stream for the master.

```
system ::  [b] -> (Work -> b -> Result) -> Int -> Process [Work] [Result]
system localList f numWorkers
    = process workList -> master # (restWork,fromWorkers)
      where restWork    = drop numWorkers workList
            fromWorkers = merge # [ worker f (localList!!i) # (workList!!i)
                                                          | i <- [0..numWorkers-1]]
```

◁

This example forms the simplest case of a system with time dependencies. More complicated systems will be investigated in Chapter 3.3.

Referential transparency. The use of `merge` inside of a function would destroy its referential transparency: the output of this function would no longer be independent of the arrival times of the merged input data.

In order to retain referential transparency on the function level, we restrict the use of `merge` and of nondeterministic processes built using `merge`: Functions can never instantiate a nondeterministic process in order to compute their result. Otherwise they would become nondeterministic functions, an undesirable feature. Note that this is the reason why `system` is defined as a process in the above example.

We distinguish between process abstractions (resp. processes) which are deterministic and ones which are (potentially) nondeterministic. A `merge` process belongs to the latter category, and every process that uses an instantiation of a potentially nondeterministic process abstraction will be regarded as potentially nondeterministic itself.

By use of this "pessimistic" estimation, the nondeterminism property is propagated and it can be inferred statically which process abstractions are potentially nondeterministic. This class of processes may only be instantiated by processes, not by functions. Equational reasoning techniques can still be applied to the remaining functional parts (see also Chapter 3.3.3).

2.6 Related Work

2.6.1 Concurrent Logic Programming

Shapiro [Sha89] identifies processes, communication, synchronization and indeterminism as the basic notions of concurrency. According to [dKC94, Chapter 3.4], systems which allow the representation of *reactive systems* are called *concurrent logic languages*. The most important programming concept underlying these languages is the process interpretation of logic programs, i.e. every goal is seen as a process, and the communication is performed by logical variables. The possible behaviour of a process is specified by so-called *guarded Horn clauses*. Among the clauses whose guard evaluates to *true*, one is selected nondeterministically. A well-known representative of this class is the language Strand[FT90].

2.6.2 Concurrent Constraint Programming

Saraswat's Approach. In [Sar93] the framework `CC/CP` is introduced. These languages entirely rely on *implicit* parallelism, since all computations are carried out in a constraint-based way. They use ask- and tell operations on a constraint store. A number of programming languages are more or less close descendants of this approach, such as **AKL**[FHJ92] and **Oz**[Smo95].

Goffin. As Eden, the language Goffin[CGKL94, CGKL98] emerged from a pure functional language and shows a two-level structure. The lower level, i.e. the *computation language*, is formed by a sequential functional language (here: Haskell). The upper level, the so-called *coordination language*, introduces and controls parallelism. In Goffin, this part is formed by simple constraints, whereas in Eden it is formed by *processes*.

2.6.3 Parallel functional approaches

There are a number of functional approaches which use annotations in order to identify expressions to be evaluated in parallel. Viewed from the expressibility point of view, these languages can however be seen as *implicitly parallel*, because the semantics is not changed by the annotations.
Para-Functional Programming[Hud91] introduces annotations for specifying both the order of evaluation (*scheduling*) and the placement of evaluations on different processors (*mapping*).
Caliban[Kel89] extends a functional language with processes, but is restricted to static process networks. Like Eden, head-strict lazy streams are used for communication.
Parallel Programming with Monads [JH93] extends monads by a `fork` primitive for the parallel execution of two computations, which combines two "processes" by returning a pair of their results. Communication in this approach is equivalent to one-value channels in Eden, so that some programs which are expressible using lazy streams cannot be realized with this monadic channel IO.
Concurrent Clean[vENPS89, NSEP91, vEPS89] is a lazy, higher-order functional language with implicit parallelism. It provides annotations for influencing the order of evaluation. It supports dynamic process creation.
Haskell + Evaluation Strategies[THLP98] is an approach defined in order to overcome the shortcomings of the par-annotations, which do not foster parallelism sufficiently. It resembles the skeleton approach in that a sequential description of the computation is 'modularly' combined with a specification of the parallel coordination.

2.6.4 Functional approaches with concurrency

Kahn and MacQueen[KM77] first proposed a model for functional processes communicating using lazy streams.
The concept of a **functional operating system**[Tur92] developed in the 1980s relies on the model of stream-based communication as well (see e.g. references in [PGF96]).
HOPSA[BDS93] is a stream-based language which is based on functional programming. The language uses fine-grained *agents* to express concurrent computations. Besides the declaration of agents, *channels* have to be declared separately by equations. A fair merge construct is present, too.

Concurrent ML [Rep91, PR97] is based on the strict functional language ML and it uses synchronous communication via typed channels. Both concurrent actors (*threads*) and channels can be generated dynamically. An *event* is an abstraction for a synchronous operation analogous to the λ-abstraction for functions. These events are first-class values. The language features a nondeterministic choice combinator for generalized selective communication.
Concurrent Haskell [PGF96] is in some sense a successor to CML, but with the differences that a lazy computation language is used and I/O is realized with monads. Synchronization is implemented by use of so-called `mvars`. They can be written more than once, but every write operation has to be preceded by a read operation, otherwise an error is raised. Attempts to read a `mvar` that has not received a value yet will block.

The language relies on shared-memory constructs and is not intended for parallel computations.
Maude [MW92] is a functional and object-oriented language. There are three types of modules: functional, system and object-oriented modules. System modules can encapsulate nonfunctional elements. Maude relies on *implicit* parallelism.
Erlang[AWV96, Wik94] is a very restricted functional language with concurrency extensions. **CD – Scheme**[Que92] defines a concurrent and distributed extension of the functional language **Scheme**. Further concurrent functional approaches are $\mathbf{8\frac{1}{2}}$[Mic96] and **ProFun**[GH96].

Process algebra-based approaches

There are a number of concurrent programming languages which are derived from process calculi. For example, **PICT**[Pie94, PT97] is based on the π-calculus and **LCS**[Ber95, Ber97] on CCS.
Facile [Tho94, GMP94, TLK97] combines ML with a concurrency model based on CCS and further higher-order and mobile extensions. **CDS***[LH97] combines the events of process algebras and the explicit processor locations of BSP with higher-order functional programming. It does not offer polymorphism and dynamic creation of processes.

2.6.5 Other related approaches

There are a number of data parallel languages, see e.g. Szymanski's book[(ed95] for an overview. The language **SISAL**[Ske91] is probably the most successful functional language for parallel numerical analysis. However, the language versions that are implemented and used seem to lack very important features of modern functional languages.
NESL [Ble96b, BCH+93] is based on ML and pursues a data-parallel approach. It supports nested structures and nested data parallelism. A further data-parallel language is [Kuc96].
FASAN[EPZ95] forms a coordination layer which can be combined with various computation languages. Its aim is especially the programming of numerical applications with imperative computation languages. It defines agents which communicate via unidirectional streams.

A language that tries to combine the advantages of SISAL and Haskell is **FSC** (Functional Scientific Computing)[Ang96]. It uses strict evaluation, a restricted form of polymorphism and a less rigorous type system. The simple functional language **SAC** (Single

Assignment C)[Sch97a, Sch96b], supports very powerful operations on arrays and restrict the functional language.

Like in Eden, in the dataflow language **pH**[NAH+95] a deterministic layer (or sublanguage) and a nondeterministic one can be distinguished. A functional core is on a first level extended with I-Structures and on a second one with M-Structures. Nondeterminism is introduced (exactly) by the M-structure extensions.

Skeleton languages. Cole[Col89] was the first to propose this idea. Nevertheless his approach is not equivalent to the ideas discussed in the following. Above all, Cole's skeletons could not be combined with each other and in this way did not correspond to higher order functions. There is a large number of different approaches, which combine imperative or declarative computation and coordination languages.

Chapter 3

Expressive Power of Eden

3.1 Different communication paradigms

As the first step towards an analysis of Eden's expressibility, we will investigate subsequently how different communication paradigms can be encoded.

Andrews[And91, p511] classifies communication paradigms with regard to the way activation is carried out and to the agent that is servicing this activation:

communication paradigm	activation	serviced by
remote procedure call	synchronous	new agent
rendezvous	synchronous	existing agent
dynamic process creation	asynchronous	new agent
asynchronous message passing	asynchronous	existing agent

This gives an abstract description of the mechanisms and their inter-relationship. In contrast to parallel imperative languages like [AO91], in Eden it is not possible to discern the mechanisms on the side of the sender from those on the side of the receiver. Nevertheless all the above communication paradigms can be expressed quite well.

Remote procedure call. Using this communication paradigm, the client has to block until its request is serviced and the service has to be provided by a process *specifically created* for this request. This synchronous invocation mechanism is also used in the object-oriented standard CORBA[Wat96]. In Eden, the respective client can be expressed by the following, where `f1` and `f2` are arbitrary functions:

```
client :: Process a  b
client = process inp -> outp
         where   outp = let ( x = serv # y ) in f1 x
                 y    = f2 inp
```

It is essential that the process instantiation `serv # y` is given as a local definition and not on the top level, because the former ensures that the process is instantiated on demand for x, which is the correct behaviour for a remote procedure call. This solution uses slightly more interleaving than the usual imperative version of remote procedure call with *call by value* parameter passing, because the process `serv` can be created before the full input is known. An (inefficient) call by reference version could be expressed by supplying the arguments as *parameters* of the process abstraction.

Rendezvous. In contrast to remote procedure call, with rendezvous the request is serviced by an already existent server process. This type of communication can be expressed in Eden by using reply channels. The client produces a request message which contains a reply channel for receiving a reply of some type `Work`. The client will block until the reply of type `Reply` arrives, which is needed for the evaluation of `result i reply`. It is assumed that there is a function `result` which yields a value of some type `Result`. The server waits for messages with reply channels to arrive and otherwise blocks.

```
client :: Process Work (Chan_name Reply, Result)
client = process inp -> workPair inp
         where workPair :: Work -> (Chan_name Reply, Result)
               workPair i = new (chan, reply) (chan, result i reply)
               result :: Work -> Reply -> Result
server :: Process [Chan_name Reply] [Reply]
server = process requests -> replies requests
         where replies :: [Chan_name Reply] -> [Reply]
               replies (c1:rest) = c1 !* reply1 par replies rest
                                   where reply1 = ...
```

Dynamic process creation. In order to create a new process which will service multiple requests (from its parent or from other processes), one can use stream-communication. A server process which processes streams is created either via a top-level definition or on demand. This process can return results and wait for more input in the future (the input must not be closed too early).

Message passing. *Asynchronous* message passing is the typical communication paradigm in Eden. The program below shows a process that is connected to a partner process via one input `from_partner` and one output `to_partner` and which exchanges with this partner one message per direction. In Chapter 3.3 we will discuss in detail in what sequence such message passing operations are carried out.

```
mp :: Process InData OutData
mp = process from_partner -> to_partner
     where to_partner  = outgoingMsg
           incomingMsg = from_partner
```

Synchronous message passing differs from the asynchronous version in that send is blocking. However, the send operation does not have to wait for a reply (i.e. a result), but only for an acknowledgement of reception. These acknowledgements can be handled in Eden by the use of reply channels. The client sends a message that contains a reply channel to the server and continues after it has received the acknowledgement via this reply channel. The real *result* is then received via a statically declared inport. This result message will probably contain a reply channel as well, which the client will use to send back a receipt.

```
syMp :: Process InData (OutData, Chan_name Ack)
syMp = process from_partner -> to_partner
       where to_partner   = new (chan, ack) (outgoingMsg, chan):wait ack
             wait Ack = ... -- continue with rest of computation
```

The receiver will answer this by:

```
answer (incomingMsg, chan) = chan !* Ack par (work incomingMsg)
work     msg = ...
```

This section has shown that all of the well-known communication paradigms mentioned above are expressible in Eden. This does of course not imply that it would be a good idea to transform existing imperative programs by replacing the code for communication directly by the code shown above. The level of abstraction underlying such a translation would be too low to produce natural Eden programs. However, this survey indicates that all essential tools are present in Eden. Thereby it gives us the freedom to study specific parallel programming problems in the following sections without trying to translate existing implementations "literally".

3.2 Transformational process systems

3.2.1 Laziness versus strictness in a parallel setting

There was a lively debate over the virtues of strict and lazy evaluation in general, as for example described in [Wad96, BJdM97, Bra95]. In some cases, such as in the case study using the pseudoknot benchmark[HFA+96] high speed required heavy use of strictness annotations. But there are also examples where lazy evaluation outperforms eager evaluation[BJdM97].

There is agreement that useful parallelism can not be obtained using *entirely* demand driven computations. A number of researchers contend that in a parallel setting it is most useful to use strict evaluation by default. [Wad96] for example states that strict evaluation in Erlang is crucial in order to be able to guarantee a specified order of evaluation. Furthermore [Hai94] argues that strictness is desirable in order to make the complexity of operations clearer.

In our view, however, the presence of concurrency or parallelism does not necessarily mean that evaluation had to be strict.

Although for a number of applications[Ang96, Re94] lazy evaluation is said to be not advantageous, this list reveals a certain preoccupation with scientific computing. This is probably an area that does indeed not benefit much from laziness (see also Chapter 9). But we do not want to restrict ourselves to this application area and for general applications, the added flexibility is very important for prototyping. The paper [GSWZ95] even describes a scientific program where laziness is said to be essential.

Limited Laziness. In the context of parallel systems, there exist the following approaches which keep lazy evaluation as a basis and introduce certain modifications that prevent the language from becoming "overly lazy":

1. The programmer explicitly indicates parts of a parallel program which are to be evaluated to normal form. This can e.g. be done using annotations [vENPS89] or evaluation strategies[THLP98].

2. The language defines suitable defaults that reflect the expected structure of a parallel program. Such a convention for the control of demand has for example been adopted in Caliban[Kel89], where head-strict lazy lists are used for communication.

We opt for the second alternative and introduce the conventions shown in Chapter 2.3. In later chapters we will show how this interacts with specific mechanisms to control speculative computations. In the following we will analyze the implications of the channel types for Eden programs.

3.2.2 The role of channel types

With **streams**, the behaviour observable by the user is that of a *one-by-one* transmission of the elements. Consequently, the programmer can rely on the fact that systems with mutual recursion over lazy lists work without any change[1]. Nonetheless, the implementation will carry out optimizations whenever this is safe, i.e. when the normal forms of more than one list element are available at the time of sending. In Chapter 6.1.1 we explain in detail the benefits of streams from an implementation point of view.

On the other hand, with **one-value channels**, the behaviour observable by the user is that of the transmission of *one message*. Again, message passing will be implemented in a way determined by the system, dependent on available buffer space and speed of production and transmission of the data.

This means, for data types other than lists, transmission does not support laziness by default, i.e. output data is evaluated to normal form by the sender and is transmitted fully to the receiver. If this behaviour were not appropriate for a particular application, a special representation of the data structure(s) could be adopted: The programmer can either resort to a stream of tree-elements or to a special definition like the one shown in the next example.

Implementing lazy data structures. One can easily define one's own lazy data structure using specific joints with unique labels, which can be requested using dynamic reply channels. The producer process evaluates and outputs an initial part of the data. For further parts of the data, "laziness" is employed in two ways:

1. The *transmission* from producer to consumer is demand-driven.

2. The producer *evaluates* output data only on demand by the consumer.

Example 3.1

In the following we present a lazy tree as an example of a general lazy data structure. In this case, the tree is transmitted node by node upon request. The producer uses a function `iniNF` in order to produce an initial part (here: one node) of the data structure, which is sent to the consumer directly. The definition of the data type `Tree` to be transmitted is extended by an additional data constructor `Joint` which marks a point where the evaluation and/or transmission has been interrupted. It takes as argument a unique label, which can be used by consumers wanting to access the subexpression 'cut off' at this position. A consumer which needs the information belonging to `label` sends a message `SendMore label chan`, which requests the respective contents to be sent across the dynamic channel `chan`.

The producer uses an arbitrary unique addressing scheme for assigning labels to the joints. In our example, we assume the inscriptions of the nodes to be unique and we use strings which contain this number, followed by `0` for the left and `1` for the right subtree.

[1] If general message passing were used, choosing a too great message size could result in deadlocks.

On interrupting the evaluation, the leftover substructures are collected in a list structured dictionary which pairs labels and substructures. The function `handleReq` looks up the requested substructures, evaluates it to the form defined by `iniNF` and sends the result across the dynamic channel enclosed in the request message.

```
data Tree  = Node Int Tree Tree | Empty | Joint Label     deriving (Eq, Show)
data Req   = SendMore Label (Chan_name Tree)              deriving (Eq, Show)
type Label = String

producer :: Process [Req] [Tree]
producer = process reqs -> sendLazyTree reqs
           where
           sendLazyTree inp = iniTree:(handleReq inp dict)
                                   where (iniTree, dict) = iniNF myTree
                                         myTree = ...
           iniNF :: Tree -> (Tree, [(Label,Tree)])
           iniNF Empty = (Empty, [])
           iniNF (Node x left right) = (Node x (Joint labelL) (Joint labelR),
                                        [(labelL, left), (labelR, right)])
                                       where labelL = (show x) ++ ",0"
                                             labelR = (show x) ++ ",1"

           handleReq :: [Req] -> [(Label, Tree)] -> [Tree]
           handleReq [] _ = []
           handleReq ((SendMore i chan):rest) labelDict
             = chan !* initree_i par (handleReq rest (labelDict ++ newDict))
               where (initree_i, newDict) = iniNF tree_i
                     tree_i = selectElem i labelDict
                     selectElem i [] = error "Label not found"
                     selectElem i ((j,tj):rest) = if i == j
                                                  then tj
                                                  else selectElem i rest
```

Accordingly, the consumer uses a function `consumeTree` in order to re-assemble the tree transmitted by the producer. The evaluation of the result is performed in a function `makeResult`, which determines which parts of the input tree are required. Note the interaction between lazy evaluation in `makeResult` and consumption of input in `consumeTree`: requests are only sent for the demanded subtrees.

```
consumer :: Process [Tree] (Result, [Req])
consumer = process inTree -> makeResult (consumeTree (head inTree))
           where
           consumeTree Empty         = (Empty, [])
           consumeTree (Node x l r)  = (Node x getL getR, reqL ++ reqR)
                                       where (getL, reqL) = consumeTree l
                                             (getR, reqR) = consumeTree r
           consumeTree (Joint label) = new (chan, cont)
                                       (result,(SendMore label chan):req)
                                       where
                                       (result, req) = consumeTree cont
           makeResult :: Tree -> Result
```

```
system = consumerResult
         where (consumerResult, reqs) = consumer # producerOut
               producerOut            = producer # reqs
```

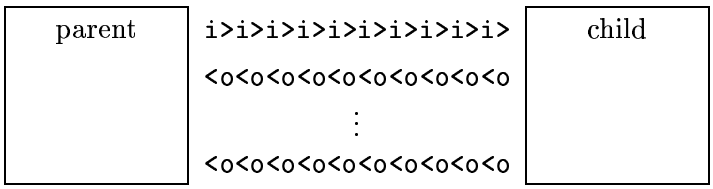
◁

The above example could easily be modified so that larger parts of the data structure are transmitted in one message. A whole library of useful lazy data structures, together with suitable functions for evaluation, transmission and consumption could be provided. Using these functions, programs could be developed in a modular way, using a flexible *decomposition - transmission – consumption* – scheme.

The role of channel structures. Another form of special support for laziness is provided by channel structures. To start with, we show a simplistic example of two communicating processes, that exposes the benefits of this feature very clearly.

Example 3.2

Consider 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
```

For this example, it is 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. ◁

In our paper[BKL97b] we show that this example can also be handled without channel structures, namely by constructing the required channel list indirectly by introducing a list of dynamic reply channels. This emulation is interesting in order to make clear what can be achieved using dynamic channels. The mechanism used resembles the one used in Example 3.1.

Channel structures provide *structured* access to input data and thereby provide added support for lazy consumption of input. This mechanism works irrespective of the number of processes that is generating this input. In the example above, the whole channel structure was produced by one process. In the context of skeletons, it is frequently the case that the substructures are generated by several processes, cf. Chapter 3.2.4.

3.2.3 Communication topologies and data distribution

In the following, we will discuss the importance of topology information for parallel programming and possible ways to express this in Eden.

Distributed data structures. A very important difference between Eden and e.g. implicitly parallel functional approaches is that Eden processes define a combination of functions and input data. This means, that by describing the topology of a process network, one at the same time describes the distribution of data over this network. Consequently, approaches to a description of the distribution of data (see e.g. [BK95, SPOP97, PS97]) can be expressed in Eden. We will come back to this aspect in Chapter 9.3, where we propose a flexible and abstract distribution of data objects.

Topology information versus placement information. In Chapter 2 we have said that arbitrary topologies can be expressed in Eden. In order to estimate correctly the role of topology information, we would like to emphasize that we are speaking about a logical distribution of processes and data, not of the placement on a physical machine. The latter aspect will be dealt with in Chapter 6.1.2.

This corresponds to the approach[Pan96], which postulates that information about communication topologies should be expressible in parallel programming languages. In this article, Panangaden argues that there should be an *abstract* notion for the distribution of computations and data, which neither leaves the placement completely implicit nor requires low-level definitions of a static location for processes or data.

3.2.4 Skeletons for stable process networks

In the following, we will present an example that shows how useful explicit and stable process networks are.

Example 3.3

A binary fully balanced process tree is defined using process abstractions `nodeAgent` and `leafAgent` for the inner nodes and leaves respectively. The node agents have three input and output channels: one input from the top and two from the bottom, one output to the top and two outputs to the successors. The leaf agents have one input from and one output to the ancestor. The depth of the process tree is given by the integer parameter `d`. The whole system transforms an `input` of some type `a` to output of some type `b`.

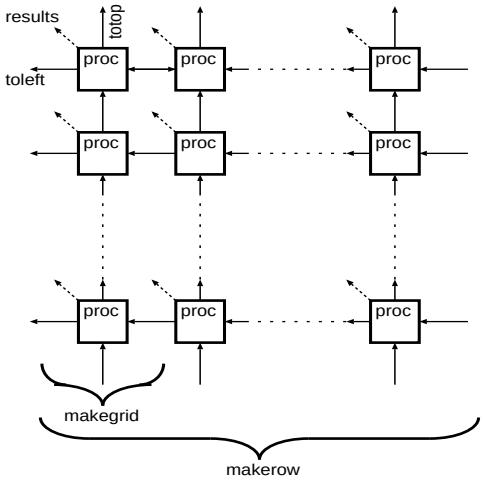


Figure 3.1: Process system generated by `grid`

```
tree :: Process (a,b,b) (a,a,b)  -> Process a b -> Int -> a      -> b
tree    nodeAgent                   leafAgent      d      input
 = if d=1 then leafAgent # input
   else toTop
   where
    (toLeft,toRight,toTop) = nodeAgent # (input,fromLeft,fromRight)
    fromLeft  = tree nodeAgent leafAgent (d-1) toLeft
    fromRight = tree nodeAgent leafAgent (d-1) toRight
```

The tree skeleton completely specifies the overall process network that underlies the parallel bitonic sort algorithm presented in [BL98]. This network is capable of sorting a *stream of input lists*. The input lists must have at least 2^d elements where d is the depth of the tree. The parameter d of `tree` determines the number and the granularity of the generated processes. In the paper mentioned above, this process system is shown to be superior to an implicitly parallel version with respect to efficiency. The process system of this Eden version is kept stable, whereas in the version with implicit parallelism processes with identical tasks are destroyed and re-created several times. This aspect will be discussed further in Chapter 7.

◁

Example 3.4

(general grid topology). The following function defines a grid process system, the dimension of which is determined by a matrix of input values passed as a parameter. The matrix is given as a list of rows. The process abstraction `proc` for the node processes is passed as a second parameter. Each node is parameterized by its position within the grid and the input value out of the input matrix corresponding to this position. Every node has two input stream channels and three output channels, two of which are streams. Accordingly, the global process structure has two lists of input and output stream channels, one input and one output channel in each row and column of the grid, and one more output channel providing a result matrix.

The topology is generated by the function `makegrid` which traverses the list of rows and calls for each row the function `makerow` which creates the processes within the rows. The communication topology resulting from `grid` is the one shown in Figure 3.1.

```
grid :: [[a]] -> (Int -> Int -> a -> Process ([b],[c]) (d, [b],[c]) )
        -> Process ([[<b>]],[[<c>]]) ([[<d>]],[[<b>]],[[<c>]])
grid matrix proc
  = process (fromRight, fromBot) -> (results, toLeft, toTop)
    where (results,toLeft,toTop) = makegrid fromRight fromBot matrix 1 1
          -- split into lists for the columns
          makegrid rs bs []         i j = ([],[],bs)
          makegrid rs bs (row:rows) i j
             = (resRow:resRest,l:ls,ts)
               where (resRest,ls,midBs) = makegrid (tail rs) bs rows (i+1) j
                     (resRow, l, ts   ) = makerow  (head rs) midBs row i j
          -- generate processes in the respective row
          makerow r bs []          i j = ([],r,[])
          makerow r bs (val:rVals) i j
             = (res:resList,l,t:ts)
                where (res,    l,   t ) = proc i j val # (midL,(head bs))
                      (resList,midL,ts) = makerow r (tail bs) rVals i (j+1)
```

◁

In the following, we will present another abstraction which relies on channel structures.

Example 3.5

It is straightforward to use the above grid topology to define a torus topology:

```
torus :: [[a]]  ->  (Int -> Int -> a -> Process ([b], [c]) (d, [b],[c]))
          -> Process () [[<d>]]
torus matrix proc = process  () -> outmatrix
                    where (outmatrix,row,col) = grid matrix proc # (row,col)
```

Gentleman's parallel matrix multiplication (see e.g. [Qui94]) can then be specified as follows. It is assumed that the square matrices `a` and `b` of dimension `n` are stored in the rotated form required by this algorithm. The multiplication can then be carried out in `n` steps, each of which consisting of one multiplication and addition and the sending resp. receiving of one message per communication partner.

```
matmult :: Num a => Int -> [[a]] -> [[a]] -> [[a]]
matmult n a b = torus ab (multproc n) # ()
                where
                ab = map (uncurry zip) (zip a b) -- matrix of pairs
                multproc n i j (aij,bij)
                  = process (inRow, inCol) -> (cij, outRow, outCol)
                    where cij    = foldr (+) 0 (zipWith (*) outRow outCol)
                          outRow = aij : (take (n-1) inRow)
                          outCol = bij : (take (n-1) inCol)
```

◁

In the previous discussion, we have given examples of typical skeletons. Other ones like pipe, farm or ring skeletons can be expressed straightforwardly. The following features of Eden proved very valuable for skeleton-based program development:

1. In contrast to many other approaches, programming with skeletons in Eden still allows *dynamic process creation*, so that the process systems generated with skeletons can be optimally adapted to the evolving workload.

2. Topologies can be described, which opens up the possibility for an optimized implementation (even platform specific).

3. Channel structures provide special support for flexible strucures.

Additionally, the paper[GPP96] presents skeletons for reactive systems.

3.3 Nondeterminism and time-dependencies

In this section, we will present a number of reactive example problems ordered with respect to the 'programming complexity' in Eden. In particular, we will revise the existing features in Eden with regard to the role they play for certain classes of examples.

The management of demand for output described in Chapter 2.3 implies that reactive systems are programmed in a way which is completely different from an imperative representation. For every process, there has to be some output and the body of the process has to be organized in a way so that the sequence of steps is directed by this output.

Every process tries to perform output operations as soon as possible, i.e. as soon as the data is available. Input will be read according to demand. This sequence does not always correspond to the one intended by the programmer.

3.3.1 Simple processes with one communication partner

The simplest case of time-dependencies is encountered when a process communicates with one partner. Time dependencies between "final" results and "intermediate" requests can be enforced easily with the help of dynamic channels (cf. e.g. the master-worker example 2.2 or also the bank example 10.1).

Strictly speaking, the fact which makes this class of reactive problems easy to handle is not the existence of only one *communication partner*, but the existence of only one output. If there is an requirement as to the sequence in which request and result messages are sent, this can trivially be enforced by placing the message in the same output stream and impose the desired order on them. The same situation could as well be represented by producing a *pair* of output streams: one for the results and one for the requests.

3.3.2 Processes with more than one communication partner

The above class of problems could be represented in Eden straightforwardly. If a process however has to communicate with multiple partners, as for example with a master process reading its results and a server process supplying it with information needed to produce the results, the situation becomes more involved. In the following we will present a very simple example so as to expose clearly the fundamental difficulties.

Example 3.6

A client receives from its master process a stream of work items. In order to handle them, it has to request information from a server process. Note that the following process abstraction does not send the requests when replies are needed by the evaluation in function `work`. The requests are sent spontaneously, because in the program below they are independent of the computation of the results.

```
client :: Process ([Reply],[Work]) ([Request],[Result])
client = process (replies,inp) -> (requests, work replies inp)
         where
         requests                          = Request:requests
         work replies      []              = []
         work (ri:repRest) ((Item i):iRest) = (result ri i):(work repRest iRest)
         result reply item                 = ...
```

◁

This is a problem frequently encountered in concurrent declarative languages: the survey [Tho93] reports that the declarative approaches invariably failed to solve this problem. The above kind of synchronization problem makes the programmer want to restrict the use of threads or specify the order of actions explicitly by special language constructs.

Restriction of threads? Although concurrent threads can lead to undesirable and unexpected behaviour in situations like the ones mentioned above, it is important to note that this effect is inherent to the approach.

Simply restricting the number of threads to one and evaluating independent outputs of a process by *one thread* would be useless, because then the problem of fairness were encountered. The question is rather if it is useful to have multiple independent *outputs*, not if it is useful to have multiple threads for multiple outputs. Many processes directly correspond to the evaluation of one computationally intensive expression and therefore they return only one output to their parent. For applications which rely on multiple independent outputs, there are the above mentioned ways to overcome this.

Note that also every process instantiation leads to new outputs for the parent process. This situation however usually does not give rise to synchronization problems, because only data dependencies are present. The above discussion rather applies to multiple outputs in the *statically declared* interface.

Explicit synchronization constructs: ask and tell. Synchronization constructs could be used in order to block a concurrent thread until certain other events have occurred. In this way, the asynchronous processing of independent threads *within* one process could be controlled by specifying dependencies explicitly.

Note that special constructs for inter-process synchronization are not needed in a distributed system. The synchronization of different processes can be implemented by use of message passing.

One of the concepts applicable for the synchronization of threads is the use of semaphores[BLO94]. In this case, the P and V operations take two arguments, one being the semaphore and the other the expression depending on it. A related mechanism on the same level of abstraction is the introduction of mutable variables like the MVars in Concurrent Haskell[PGF96].

Alternatively to this mechanism which would impose on Eden programs an imperative style, ask and tell operations could be introduced. They differ from semaphores in that not temporary states, but declarative expressions are stored.

Ask and tell operations on a local store. In contrast with the original concurrent constraint approach developed in [Sar93], here every process should be equipped with a *local* constraint store. This suffices to implement local synchronization operations and can be implemented efficiently on a distributed memory computer.

tell(k) is a nonblocking operation that adds (the normal form of) an object k of a type *Store* to the local constraint store of the process. *ask*(k) performs a lookup in the local constraint store. If object k is present, the operation succeeds, otherwise it blocks. Consequently, this could be used to implement waiting for the evaluation of an expression in the following way:

```
client :: Process ([Reply],[Work]) ([Request],[Result])
client = process (replies,inp) -> (requests 0, work replies inp 0)
         where
         requests k = (ask k `seq` Request):(requests (k+1))
         work replies      []               k = []
         work (ri:repRest) ((Item i):iRest) k
              = tell k `seq`
                (result ri i):(work repRest iRest (k+1))
         result reply item = ...
```

No consumption. It is essential to work with a monotonically growing store, because otherwise nondeterminism would be introduced in an uncontrolled way: If objects written to the store could be eliminated again, the behaviour of the following small process would depend on the sequence in which the two threads were scheduled:

```
switch = process inp -> (out1, out2)
         where out1 = expr `seq` tell Complete `seq` ask Complete `seq` inp
               out2 =                                ask Complete `seq` inp
```

This kind of synchronization mechanism is weaker than semaphores, because mutual exclusion cannot be expressed. In Eden processes, mutual exclusion problems are not encountered, but only the simpler type shown above.

Ask and tell can in principle be simulated by message passing. Ask is equivalent to an attempt to receive from an additional synchronization channel. Tell is equivalent to a send operation that is directed to this channel. That is to say, the synchronization channel connects a process to itself. If more than one thread perform tell-operations, a separate channel has to be introduced for each of them. Then **merge** must be used to transform the list of these outputs into one input. In contrast to this, ask operations can be used by multiple threads without special treatment, because inputs are visible to all threads. Note that the representation by the contents of an input trivially implements the non-consuming demand control store described above.

Based on this representation, a functional implementation of **ask** is trivial: **ask** k searches the synchronization input for object k and waits for it to appear if it is not found in the contents received up to this time. If the end of the input is reached, failure is raised.

As the tell-output is independent of the other outputs, the representation of tell is exactly as complicated as solving the problem of multiple outputs directly. The chief

benefit of such a solution is that the handling of synchronization constraints could be encapsulated, for instance inside a special monad.

In general, such explicit synchronization constructs should only be introduced if they turn out to be vital for a considerable range of application programs and if the consequences for the semantics of the language have been studied in detail. In the version of Eden presented here, they are not adopted. In the following we will present an alternative and thereby illustrate that they are not strictly necessary.

Implicit synchronization Synchronization requirements can also be expressed without additional language constructs, particularly by working with dynamic channels and data dependencies.

Example 3.7

In the following, we will present an alternative solution for the client that fulfils the synchronicity requirement that requests to the server only be sent when they are needed by the main computation. This is achieved by servicing outports using the same computation, i.e. not in *independent* subexpressions.

```
client :: Process ([Reply],[Work]) ([Request],[Result])
client = process (replies,inp) -> work2 replies inp
         where
         work2 replies      []              = ( [], [] )
         work2 (ri:repRest) (Item i):iRest)
              = ( Request:reqRest, (result ri i):wRest)
                where (reqRest, wRest) = work2 repRest iRest
         result reply item = ...
```

◁

In the above situation, the request can be released at the *beginning* of the evaluation of a new item. The solution can easily be modified to delay the sending of some output until the *end* of some computation[2], either by using `seq` or by bundling the main computation in one thread that produces a stream with all outputs, which afterwards is distributed by a filter function (see example 3.8 below).

Example 3.8

We again consider the client - server example and use the methodology with one stream of different messages generated by a function `work1`, which is achieved by use of a special data type `Msg`. The function `distrib` postprocesses the stream of messages generated by `work1` so that the ones starting with constructor `M` are transmitted to the master and the ones starting with `S` to the server. In this version, dynamic channels are used for the replies. The mechanisms can be used interchangably in this case.

```
client :: Process [Work] ([Request],[Result])
client = process inp -> distrib (work1 inp)
         where
         work1 []                    = []
```

[2]Note that the mere use of tuples of streams does not meet this requirement: with two outputs `simpleMsgs` and `complexMsgs` and a definition `(simpleMsgs, complexMsgs) = (s1:rest1, c1:rest2)`, `s1` can be sent to `simpleMsgs` while the computation of `c1` is still under way.

```
        work1 ((Item i):rest)     = new (chan, reply)
                                        (S chan i):
                                        (M (result i reply)) : (work1 rest)
        result i reply = ...

        distrib [] = ([],[])
        distrib ((S contents):rest) = (contents:toServ, toMast)
                                      where (toServ, toMast) = distrib rest
        distrib ((M contents):rest) = (toServ, contents:toMast)
                                      where (toServ, toMast) = distrib rest
data Msg  a   = M a | S (Channel_name Reply) Int
```

◁

Concluding remarks. By use of the above method, one trivially solves the synchronization problem by bundling the computation of multiple outputs into one thread. Although there may be several threads present, only one is doing the work. This kind of 'expressibility' however comes at a considerable expense: logically unrelated functions for the various results have to be combined into one big function that contains all of them. This is some kind of combinatorical explosion which can lead to inelegant programs.

Apart from this programming penalty, there will of course also be a performance penalty resulting from this artificial synchronization. Note that in this approach, one fails to express the requirement that certain local steps be synchronized (e.g. the sending of a request to the server exactly in the moment a new value is needed) and synchronizes all the steps instead. This means in some cases, that computations and communications cannot be overlapped properly.

On the other hand, synchronization mechanisms were found to be not strictly necessary. Consequently, they are not introduced for the sake of a 'lean design'.

Explicit representation of time. The above class of synchronization problems still represents the case that an *abstract* notion of time is used, i.e. one action has to be delayed until some other is started or completed. A more explicit handling of time would be required if processes had to wait exactly for a *specified amount of time*[3].

3.3.3 The role of nondeterminism

In this section, we will discuss where nondeterminism is beneficial and in what Eden programs it can be found. Nondeterminism in parallel languages can be seen as the consequence of combining parallelism and state. Consequently it can be suppressed by localizing[4] the use of state (C. Kirkham[DAB+95]).

Contrary to the view taken in sequential functional languages, the presence of nondeterminism can have very positive effects in a parallel setting. It alleviates the elimination of unnecessary details by making "irrelevant differences" invisible. In this way, it supports the natural and abstract representation of computations that **produce determinate results without being deterministic**[DAB+95, CK94]. This is the "safe" use of

[3]A special timer process would be only of limited use because asynchronous communication is used.
[4]Note: the ask and tell operations proposed before also use an only *local* constraint store.

nondeterminism that helps to make computations flexible and efficient without affecting their result. But apart from this form, there are also other forms which compromise safety or clarity. In the following we will investigate whether only safe nondeterminism or also unsafe nondeterminism can be found in Eden programs.

Of course, the decision what is regarded as nondeterministic, depends on the notion of *equivalence* used. In our context, we identify the following two forms of (unsafe) nondeterminism:

Non-referentially transparent behaviour: A program is determinate, i.e. whenever it generates a result, this result is the same. For some expressions, however, no result is produced, although the results of all subexpressions are acceptable arguments.

Non-determinate result: Different runs of a program for the same input can generate different results.

We will now study which of these forms can arise from the use of the two non-functional constructs of Eden, namely `merge` and dynamic reply channels.

Merge

The nondeterministic process abstraction `merge` introduces only the second of the above forms. Clearly, the first is impossible because no use of merge can result in run-time failure.

On the other hand, with merge it is trivial to write a process with a non-determinate result. This process generates simply the output `head (merge # [[True],[False]])`. Note, however, that this process can easily be identified as potentially nondeterministic (cf. Chapter 2.5). Turner[Tur92] states that with **merge**, every form of nondeterminism is expressible.

Dynamic reply channels

In the expression `c !* e1 par e2`, the assignment[5] of `e1` to the dynamic reply channel `c` forms a side effect and `e2` the result. In the following we will discuss whether this fact is a source of nondeterminism in the sense described above.

Example 3.9

Below you see a program that incurs the well-known problems with laziness and side-effects. Due to insufficient sharing there is an attempt to connect to a reply channel more than once, resulting in a runtime error for applications of `f_err`, but not for applications of `f_ok`:

```
f_ok, f_err :: Chan_name Int -> [Int]
f_ok  cName = let x = cName !* 0 par 1 in [ x                | i <- [1..] ]
f_err cName =                             [ cName !* 0 par 1 | i <- [1..] ]
```

The two functions look very similar, but one doesn't behave in the desired way, due to the inappropriate use of side effects. There is no indeterminacy in the behaviour of these two functions, though. ◁

[5]Note that this is not a send-operation, but the definition of the output to be sent across the channel. Sending is implicit, as usual.

In the following, we will show an example which helps to investigate if programs with dynamic channels can produce nondeterminate results.

Example 3.10

Consider the following "pathological" example: A process `parent` instantiates two processes `child 1` and `child 2`. Both these processes receive the same dynamic channel name `cName` as input. In dependence of their second input, each of them decides whether to use this channel in order to send its id to the parent.

```
parent :: Process InData [Int]
parent = process inp -> out
         where out = new (cName, cCont) cCont:(( child 1 # (cName,inp1))
                                              ++ ( child 2 # (cName,inp2)))
               inp1, inp2 :: InData
child :: Int -> Process (Chan_name Int, InData) [Int]
child myId = process (cName, inp) -> out
             where out = if (decide inp) then cName !* myId par []
                                         else                   []
                   decide :: InData -> Bool
```

For general inputs, the following outcomes are possible:

- `child 1` and `child 2` both try to connect to `cName`, resulting in a runtime error due to the repeated use of the dynamic channel.
- only `child 1` connects, `parent` generates `[1]` as its result
- only `child 2` connects, `parent` generates `[2]` as its result
- neither `child 1` nor `child 2` connect, so that `parent` blocks because it does not receive input from `cCont`.

Nevertheless, this program will not show varying behaviour for the *same input*. The reaction of a child process is a function of its input, and its input is specified functionally in the parent process[6]. ◁

The above example shows the lack of programming discipline: the use of one dynamic channel as the input to two different processes violates the so-called *pass once* requirement (cf. Chapter 2.4). This requirement has to be kept in mind by the programmer as a guideline. An exhaustive static check of this condition is neither desirable nor feasible[7].

In summary, for dynamic reply channels we found examples which did not show their runtime behaviour as clearly as it is expected from declarative programs, but we did not find the two forms of nondeterminism considered.

Analysis of properties. We have identified two sources of time-dependent behaviour. The process abstraction **merge** is a true source of nondeterminism. Dynamic reply channels are an obstacle to reasoning because non-local definitions can heavily influence the

[6]If `parent` used nondeterministic processes to generate `inp1` and `inp2`, this (and not the use of dynamic channels) would be the reason for `parent` generating a nondeterministic result.

[7]Such analyses would inevitably preclude programs of the above type, even if it were guaranteed that always exactly one of the child processes used the channel.

whole system. This could e.g. be done by defining a separate class of processes for processes that use dynamic channels. This property is visible statically, like the use of **merge**.

The examples shown above can serve to illustrate the possibilities and limitations of the non-functional constructs in Eden. In order to show that e.g. dynamic reply channels can not produce nondeterminate results, of course, only a formal proof would be conclusive. This would however be beyond the scope of this chapter.

These programs are however rather contrived specimens of wilful program design, which are unlikely to be found in realistic programs. In contrast to the ones above, practical programs will be easy to analyze, because the non-functional extensions are used only when needed.

3.4 Summary and outlook

3.4.1 Eden 1.0

Expressibility. We have shown that Eden cannot only express deterministic computations, but also reactive computations with *abstract time dependencies.*

Programming efficiency. In the version of Eden described here, priority has been given to a *lean* design and to the use of *functional* mechanisms. For practical use in real-world applications, of course a study of the theoretical expressibility of Eden does not suffice. Based on larger applications, we will be able to answer critically the question which classes of problems can be modelled in a natural way, and which should better be handled by extended mechanisms in the language, such as the ones sketched below. This question will be discussed further in Chapter 9.5.

3.4.2 Possible extensions

I / O model. Input/Output for a long time has been one of the obstacles to a broad acceptance of functional languages in practice (see [Wad97] for a general discussion). These problems even aggravate in a parallel setting, where I/O is not only hard to handle, but also very expensive[8].

In Eden, we adopt the view taken in most *parallel* programming languages: output is performed by one main process that runs on a main processing element (PE). Additionally, processes on other PEs could be allowed to send messages to the standard output of this main PE, which then would be preceded by the id of the sender. Input is only allowed from the main process.

The integration of more powerful approaches which are also suitable for programming graphical user interfaces, remain for further work. There are a number of such approaches, based on monads, e.g. GUI interfaces [SSV96, HC95, Sch96a, EH97].

Support for foreign-language interfaces. The possibilities offered by Glasgow Haskell's C interface *Green Card*[NP96] can be used without change. A special application of

[8]Supercomputers are humourously described as devices that transform CPU-bound problems into I/O-bound ones.

Green Card is the MPI+Haskell prototype implementation of Eden described in Chapter 7. Although interfaces to languages with side-effects always introduce considerable dangers, they are very important for a language that will be used for rapid prototyping. If the power of Green Card is sufficient for the interaction of Eden processes with C+MPI processes in real-world applications, will become clear when large application programs are available.

Time-dependent behaviour. Additional "system" process abstractions like **merge** could be added if necessary. Thanks to the clear separation between such process abstraction and the functional part of Eden, such an extension would be simple and safe. Possible candidates for such extensions are a time-server and variants of **merge** that select inputs in the precise order of their arrival or that give certain input channels priority over others.

More complicated than that would be the inclusion of synchronization constructs discussed in Chapter 3.3.2. For the time being, it appears that only a small class of problems would benefit from the added expressibility of such constructs. If they were to be adopted, the resulting nondeterminism would have to be studied in more detail.

Placement on a parallel machine. We intend to define coarse-grained processes which live as long as possible. Under these circumstances it doesn't seem very promising to assign processors to processes automatically. It would be convenient to allow scheduling annotations in a process application (cf para-functional programming [Hud91]). Our notion of a process as a set of computations running on the same processor could easily be enhanced by optional "placement information" directing the assignment to physical processors. Following the terminology of [CGKL94], such an approach could be classified as "fully explicit" in contrast to the semi-implicit parallelism of Goffin.

Generalizing streams. For data types other than lists, no special convention for 'lazy transmission' is defined, but this would be an interesting area for future work.

Formal methods. In Eden, potentially nondeterministic process abstractions can be statically distinguished from deterministic ones. In this chapter, we have investigated the properties of isolated constructs. But knowledge about the properties of whole Eden programs at the moment can only be inferred by the programmer.

Special formal analysis techniques will be devised in the future. This analysis has to be accompanied by an inquiry into the implications of nondeterminism for *efficiency*.

A further interesting point for the analysis of program properties would be to prove freedom of deadlocks.

Chapter 4

Formal Treatment of Eden

4.1 The syntax of Eden

Lexical syntax

The lexical syntax of Haskell can be found in [PH97, App. B.2]. For the syntax of Eden, only the reserved identifiers `par` and `process` and the symbol `!*` have to be added[1].

Context-free syntax

A full context-free syntax of Haskell can be found in [PH97, App. B.4]. For Eden, the following extensions are required:

expr	$\rightarrow$	`process` $(in_1, \ldots, in_n)$ `->` $(out_1, \ldots, out_n)$	process abstraction
	$\mid$	p `#` $(in_1, \ldots, in_n)$	process instantiation
	$\mid$	c `!*` $expr_1$ `par` $expr_2$	dynamic channel use
	$\mid$	`new` (cn, cc) $expr$	dynamic channel creation

If the grammar is extended in this simple way, the following conditions have to be enforced by additional semantic properties:

- Functions may not contain instantiations of nondeterministic processes.
- Every dynamic channel may only be used once.
- In contrast to λ-abstractions in Haskell, the inputs of a process abstraction are visible in the `where`-definitions.

4.2 Kernel Eden

In order to simplify the operational semantics of Eden (see Chapter 5), we introduce a small kernel language called **Kernel Eden**. Every Eden program can be translated into this language by straightforward transformations like pattern matching compiling or λ -lifting.

[1] Note that symbol `->` is already present.

This section contains a formal specification of the syntactic domains of Kernel Eden: type constructors, type variables and types, basic operations and constructors, variables and function symbols, expressions and scripts.

4.2.1 Types, predefined operations, and data constructors

Eden adopts the Hindley/Milner polymorphic type system, but extends it by types for process abstractions and dynamic reply channels.

- Let $\Theta = \bigcup_{n \in \mathbb{N}} \Theta^n$ be a ranked alphabet of type constructors with $\rightarrow \in \Theta^2$, $[.]$, $\mathit{Chan\_name} \in \Theta^1$, $\mathit{Bool} \in \Theta^0$ and
- TVar be a set of type variables which are used in the definition of polymorphic types.

The set $T_\Theta(\mathit{TVar})$ of Θ terms over TVar is given by:

- $\mathit{TVar} \subseteq T_\Theta(\mathit{TVar})$
- If $\sigma \in \Theta^n$, $\tau_1, \ldots, \tau_n \in T_\Theta(\mathit{TVar})$, then $(\sigma\ \tau_1 \ldots \tau_n) \in T_\Theta(\mathit{TVar})$

The set Types of types then contains the elements of $T_\Theta(\mathit{TVar})$, extended by composite types used to indicate process abstractions:

- $\mathit{TVar} \subseteq \mathit{Types}$
- If $\sigma \in \Theta^n$, $\tau_1, \ldots, \tau_n \in \mathit{Types}$, then $(\sigma\ \tau_1 \ldots \tau_n) \in \mathit{Types}$.
- If $\tau_1, \ldots, \tau_m, \tilde{\tau}_1, \ldots, \tilde{\tau}_n \in \mathit{Types}$ with $m \geq 0$, $n > 0$ then $\mathit{Process}\ (\tau_1, \ldots, \tau_m)\ (\tilde{\tau}_1, \ldots, \tilde{\tau}_n) \in \mathit{Types}$.

We use infix notation for the type construction $\rightarrow$ and denote the type of lists over τ by $[\tau]$. The type constructor $\mathit{Chan\_name}$ indicates dynamically created channels. *Monomorphic* types do not contain type variables. A *type substitution* is a finite mapping $\vartheta : \mathit{TVar} \rightarrow \mathit{Types}$. A type τ' is called a *type instance* of τ: $\tau' \leq \tau$, if there exists a type substitution ϑ with $\tau' = \tau\vartheta$.

Notation: Hence forth we will use the symbol $\oplus$ to represent the union of disjoint sets.

Eden's *signature* is a quadruple $\Sigma = \langle \Theta, \mathit{TVar}, \Omega, \Gamma \rangle$, where

- $\Omega = \bigoplus \langle \Omega^{s_0 \rightarrow \ldots \rightarrow s_n} \mid s_i \in \mathit{Types}, 0 \leq i \leq n \rangle$ denotes the set of *basic (predefined) operations (functions).*
- $\Gamma = \bigoplus \langle \Gamma^{s_1 \rightarrow \ldots \rightarrow s_m \rightarrow (\sigma\ \alpha_1 \ldots \alpha_n)} \mid s_j \in \mathit{Types}, \sigma \in \Theta^n,$ $\mathit{tvars}(s_j) \subseteq \{\alpha_1, \ldots, \alpha_n\}, (1 \leq j \leq m, n \geq 0) \rangle$ is a finite set of *constructor symbols*, where *tvars(t)* yields the type variables occurring in type t.

Let $\Gamma^{(\sigma)} := \bigoplus_{s_1,\ldots,s_m \in \mathit{Types}} \Gamma^{s_1 \to \ldots \to s_m \to (\sigma\ \alpha_1 \ldots \alpha_n)}$ $(\sigma \in \Theta^n)$ the set of constructor symbols which define the algebraic structure of types constructed by $\sigma \in \Theta$.

We assume that

$$\begin{array}{ll} \Gamma^{(\mathit{bool})} = \{\mathrm{True}, \mathrm{False}\} & \text{with the nullary constructors True and False} \\ \Gamma^{([.])} = \{[\,], :\}, & \text{with the empty-list constructor } [\,] :: [a] \text{ and} \\ & \text{the binary list constructor } (:) \text{ with type } a \to [a] \to [a] \end{array}$$

A declaration of the form

$$\texttt{data}\ (\sigma\ \alpha_1 \ldots \alpha_n) = C_1\ s_{11} \ldots s_{1m_1} \mid \ldots \mid C_k\ s_{k1} \ldots s_{km_k}$$

corresponds to the definition

$$\sigma \in \Theta^n \text{ and } \Gamma^{(\sigma)} = \{C_1, \ldots, C_k\} \text{ with } C_j \in \Gamma^{s_{j1} \to \ldots \to s_{jm_j} \to (\sigma\ \alpha_1 \ldots \alpha_n)} (1 \leq j \leq k).$$

4.2.2 Expressions

Let $\mathit{Var} = \{\mathit{Var}^\tau \mid \tau \in \mathit{Types}\}$ be a set of typed variables (identifiers) for constants, functions and process abstractions.

Notation:

variables:	$x, y, x_i \ldots, c, d \ldots$
function identifiers:	$f, f_i, g, g_i \ldots$
process identifiers:	$p, p_i, q, q_i \ldots$
predefined functions:	op, op_i
constructors:	C, C_i

Expressions are purely applicative, i.e. we do not consider λ abstractions. Using λ -lifting, arbitrary λ expressions can be translated into our kernel language.

Whereas in Kernel Eden process instantiations and abstractions are only admitted in equations within scripts (see next subsection), in "full" Eden programs, process abstractions and instantiations may occur in arbitrary positions within expressions. The transformation into Kernel Eden introduces special equations for such abstractions and instantiations and replaces the inline occurrences with the left hand sides of these equations. Similarly, process abstractions with free variables are transformed so that these variables appear as explicit parameters.

Thus, the set of expressions of the functional kernel language is only extended by the declaration and use of dynamic reply channels.

In order to appropriately handle polymorphism, one has to distinguish between variables introduced as function parameters and those defined in scripts. The type of the latter may be instantiated freely, while the instantiation of the former must be fixed within the surrounding expression. We use a function $\nu : \mathit{Var} \to \{0, 1\}$ to indicate whether the type of a variable can be instantiated or not.
The set $\mathit{Exp}_\Sigma(V, \nu) = \bigoplus\langle \mathit{Exp}^\tau(V, \nu) \mid \tau \in \mathit{Types}\rangle$ of expressions with (free) variables out of $V \subseteq \mathit{Var}$ and $\nu : V \to \{0, 1\}$ as explained above, is inductively defined by:

(i) $x \in \mathit{Exp}^\tau(V, \nu)$, if $x \in V^\tau$ with $\nu(x) = 0$ **variables**

$x \in \mathit{Exp}^{\tau'}(V, \nu)$, if $x \in V^\tau$ with $\nu(x) = 1$ and $\tau' \leq \tau$ **identifiers**

(ii) $(\Omega^\tau \cup \Gamma^\tau) \subseteq Exp^{\tau'}(V,\nu)$ with $\tau' \leq \tau$ **predefined functions, constructors**

(iii) If $e' \in Exp^{\tau\to\tau'}(V,\nu)$ and $e \in Exp^\tau(V,\nu)$ then $(e'\ e) \in Exp^{\tau'}(V,\nu)$ **applications**

(v) Let $e \in Exp^\delta(V,\nu)$ with type $\delta = (\sigma\ \tau_1 \ldots \tau_n) \in Types$ and $\{C_1,\ldots,C_k\} \subseteq \Gamma^{(\sigma)}$, where $C_i \in \Gamma^{s_{i1}\to\ldots\to s_{im_i}\to\delta}$.

Let $y_{ij} \in V^{s_{ij}} \subseteq Var^{s_{ij}}$ pairwise different variables $(1 \leq j \leq m_i)$ and $e_i \in Exp^\tau(V \cup \{y_{i1},\ldots,y_{im_i}\}, \nu[y_{i1} \mapsto 0,\ldots,y_{im_i} \mapsto 0])$ $(1 \leq i \leq k)$.

Then:

$\texttt{case}\ e\ \texttt{of}\ \{(C_1\ y_{11}\ldots y_{1m_1}) \to e_1;\ldots;(C_k\ y_{k1}\ldots y_{km_k}) \to e_k\ \} \in Exp^\tau(V,\nu)$

pattern matching

(vi) Let $eqn \in Script_\Sigma(V^{\text{in}}, V^{\text{out}})$ be a script over the set of variables $V^{\text{in}} \subseteq V$ defining identifiers of the set $V^{\text{out}} \subseteq Var$ with $V^{\text{out}} \cap V = \emptyset$ and $e \in Exp^\tau(V \cup V^{\text{out}}, \nu^+)$ where $\nu^+(x) = \begin{cases} 1 & \text{if } x \in V^{\text{out}} \\ \nu(x) & \text{else} \end{cases}$.

Then: $\texttt{let}\ eqn\ \texttt{in}\ e \in Exp^\tau(V,\nu)$ **local definitions**

(vii) Let $cName \in Var \setminus V$ with type $Chan\_name\ \tau$ and $c \in Var \setminus V$ with type τ.

Let $e \in Exp^\tau(V \cup \{cName, c\}, \nu[cName \mapsto 0, c \mapsto 0])$.

Then: $\texttt{new}\ (cName, c)\ e \in Exp^\tau(V,\nu)$ **reply channel declaration**

(viii) Let $e \in Exp^\tau(V,\nu)$, $e' \in Exp^{\tau'}(V,\nu)$ and $c \in V$ with type $Chan\_name\ \tau'$. Then:

$c\ !*\ e'\ \texttt{par}\ e \in Exp^\tau(V,\nu)$ **use of reply channel**

Applications are left associative. Brackets are usually omitted: $(e\ e_1 \ldots e_n)$ corresponds to $(\ldots((e\ e_1)\ldots)e_n)$.

4.2.3 Scripts

A *script eqn* is a set of equations defining variables, functions, process abstractions and process instantiations.

Let $V = V^{\text{in}} \oplus V^{\text{out}} \subseteq Var$ and $\nu : V \to \{0,1\}$ with $\nu(x) = \begin{cases} 0 & \text{if } x \in V^{\text{in}} \\ 1 & \text{if } x \in V^{\text{out}} \end{cases}$

Then, a script $eqn \in Script_\Sigma(V^{\text{in}}, V^{\text{out}})$ over the set of variables V^{in} defining the variables and identifiers in V^{out} may contain equations of the following form

1. **variable definition:** $y = e$
   with $e \in Exp^\tau(V,\nu)$ for $y \in V^{\text{out}}$ with type τ.

2. **function definition:** $f\ x_1 \ldots x_n = e$
   with $e \in Exp^\tau(V \cup \{x_1,\ldots,x_n\}, \nu[x_1 \mapsto 0,\ldots,x_n \mapsto 0])$,
   $f \in V^{\text{out}}$ with type $\tau_1 \to \ldots \to \tau_n \to \tau$ and $x_i \in Var^{\tau_i} \setminus V$.

3. **process abstraction:** $p\ x_1 \ldots x_k =$ `process` $(in_1, \ldots, in_m)$ `->` $(out_1, \ldots, out_n)$ `where` eqn

   with $p \in V^{\text{out}}$ with type $\tilde{\tau}_1 \to \ldots \to \tilde{\tau}_k \to \mathit{Process}\ (\tau_1, \ldots, \tau_m)\ (\tau'_1, \ldots, \tau'_n)$,
   $x_j \in \mathit{Var}^{\tilde{\tau}_j} \setminus V \quad (1 \leq j \leq k)$,
   $in_i \in \mathit{Var}^{\tau_i} \setminus V \quad (1 \leq i \leq m)$,
   $out_l \in \mathit{Var}^{\tau'_l} \setminus V \quad (1 \leq l \leq n)$ pairwise disjoint and
   $eqn \in \mathit{Script}_\Sigma(\{x_1, ..., x_k, in_1, ..., in_m\}, \tilde{V}^{\text{out}})$ with $\{out_1, ..., out_n\} \subseteq \tilde{V}^{\text{out}}$.

4. **process instantiation:** $(out_1, \ldots, out_n) = p$ `#` $(input\_exp_1, \ldots, input\_exp_m)$,
   with $out_j \in V^{\text{out}}$ with type τ'_j $(1 \leq j \leq n)$, $p \in V$ with type

$$\mathit{Process}\ (\tau_1, \ldots, \tau_m)\ (\tau'_1, \ldots, \tau'_n)$$

   and $input\_exp_i \in \mathit{Exp}^{\tau_i}(V, \nu)$ $(1 \leq i \leq m)$.

Each variable in V_{out} must occur exactly once on the left hand side of an equation within the script. The set of all scripts is denoted by Script_Σ. A Kernel Eden *program* is merely a script.

4.2.4 Processes and their types

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_Σ contains all data types that can be used as types of messages in Eden programs. The set IType_Σ of *interface types* contains all data types that can occur as types of communication channels or channel structures. The definition of these sets is mutually recursive. It is based on the set of type variables TVar and the ranked alphabet of type constructors $\Theta = \cup_{i=0}^{\infty}\Theta^i$ introduced before. Then, MType_Σ and IType_Σ are inductively defined as the smallest sets that contain the following:

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

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

IType_Σ is a superset of MType_Σ, 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.

4.2.5 An example transformation

Each Eden program can easily be transformed into a Kernel Eden program. Type declarations with channel annotations will not be changed. Instead of a formal specification of this transformation we include only an example of a transformed program.

The transformation of the `sortNet` example shown in [BLOP96a, Section 2.2] yields the following Kernel Eden program:

```
sortNet :: Process [a] [a]
sortNet
  = process list -> result_list
    where result_list = sort list
          sort list
            =  case list of {
                 []     -> [];
                 (x:xs) -> case xs of {
                             []     -> [x];
                             (y:ys) -> let p  = unshuffle list
                                           left   = sortNet # (fst p)
                                           right  = sortNet # (snd p)
                                           result = merger # (left, right)
                                        in  result
                           }
               }
          unshuffle list = case list of {
                             []     -> ([],[]);
                             (x:xs) -> case xs of {
                                         []     -> ([x],[]);
                                         (y:ys) -> let p  = unshuffle ys
                                                   in (x:(fst p),y:(snd p))
                                       }
                            }
          merger :: Process ([a],[a]) [a]
          merger =  process (s1,s2) -> s
                    where s = smerge s1 s2
                          smerge l1 l2
                            = case l1 of {
                                []     -> l2;
                                (x:xs) -> case l2 of {
                                            []     -> l1;
                                            (y:ys) ->
                                            case (x <= y) of {
                                              True -> (x:smerge xs (y:ys));
                                              False-> (y:smerge (x:xs) ys)
                                            }
                                          }                              }
```

4.3 Normalized Kernel Eden

In this section, we will proceed from a program in Kernel Eden (cf. Chapter 4.1) and transform it into a normalized representation. Both the interfaces and the bodies of processes have to be normalized.

The transformation of interfaces determines the structure of the interface. For the bodies, we adopt the normal form of programs defined in [Lau93], in order to handle lazy evaluation and sharing in an appropriate way. The normalization process ensures that all variable names are unique and that only variables occur in argument positions. This means that each argument position is identified with a unique variable name which is used to reference the argument and to replace an evaluated argument expression by its result.

4.3.1 Normalization of interfaces

In the following, we will describe how the interface specification of a process abstraction is transformed into a uniform representation[2]. The sets $MType_\Sigma$ and $IType_\Sigma$ defined in Chapter 4.2.4 serve as the basis for this step.

A normal form of individual interface types is needed in order to resolve the defaults used for programming convenience and in order to have a clear representation of the interfaces for the use in the operational semantics. The normal form which will be presented below will mark every type that is to be transmitted via a separate channel with a channel annotation. Especially, every stream channel can be recognized by the pattern `<[a]>`.
The function *itype_nf* defines the normal form representation for the elements of $IType_\Sigma$:

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

A normal form of a process abstraction can be derived from the normal forms of the individual inputs and outputs using the following function:

$$
\begin{array}{l}
\mathit{ptype\_nf}\ (\ \textbf{Process}\ \ (t_{11}, \ldots, t_{1m})\ (t_{21}, \ldots, t_{2n})\ ) = \\
\quad \textbf{Process}\ (\mathit{itype\_nf}\ (t_{11}), \ldots, \mathit{itype\_nf}\ (t_{1m}))\ (\mathit{itype\_nf}\ (t_{21}), \ldots, \mathit{itype\_nf}\ (t_{2n}))
\end{array}
$$

It ignores the tuples that surround the inputs and outputs of the process abstraction, because they are part of the syntax and not tuples in the sense of data structures[3].

[2]As the specification of a type of a process abstraction is not mandatory, type inference has to be carried out prior to this transformation step.

[3]Function *ptype_nf* interprets the components of these input- and output tuples as individual channels. If a process abstraction occurs as the type of an in- or output of another process abstraction, *itype_nf* will be called, which infers that only one channel is used for the entire abstraction and ignores possible annotations in the component types. This has no effect, because these types play the role of types of

The type t of a process abstraction is said to be *in normal form*, if $ptype\_nf(t) = t$. This normal form representation will be the basis for the dynamic handling of channel structures in the operational semantics (see Chapter 5.6.1).

Note that for communication without *bypassing*, i.e. shortcutting of communication channels (cf. Chapter 5.3), always exactly one of the communicating processes will contain annotation information, namely the process that is the child of the other one.

Bypassing will be only possible if the *annotated* types of the respective channels are identical. Otherwise it would be necessary to inform one of the communication partners that its interface (which also contains a tag for the type of the channel) has to be changed (cf. Chapter 5.3.2). Alternatively, the most specific interface could be inferred, but such extensions of the transformation will not be addressed here.

Channel structures will be introduced as follows: Upon process creation (see Chapter 5.3), inputs and outputs which have types with normal form `<a>` for some type `a` will be directly represented by regular channels. All other inputs and outputs will be represented by channel structures.

4.3.2 Normalized Kernel Eden programs

In the following, we will first describe the transformation of an expression e into a normalized version e^*. Then the normalization process is defined inductively over the structure of expressions and scripts. The uniqueness of variable names has to be achieved by appropriately renaming bound variables.

Function applications of the form $(e\ e_1 \ldots e_n)$ are replaced by expressions

$$\begin{array}{lrcl} let & y_1 & = & e_1^* \\ & & \vdots & \\ & y_n & = & e_n^* \\ in & (e^*\ y_1 \ldots y_n) & & \end{array}$$

where $y_1, \ldots, y_n$ are *new* variables.

Process instantiations of the form

$$(out_1, \ldots, out_n) = pabs\ \#\ (input\_exp_1, \ldots, input\_exp_m)$$

are transformed in the following way:

1. Introduction of new variables for the input expressions:
   the equation shown above is replaced by the following set of equations within a script, where $y_1, \ldots, y_m$ are *new* variables:

$$\begin{array}{rcl} y_1 & = & input\_exp_1^* \\ & \vdots & \\ y_m & = & input\_exp_m^* \\ (out_1, \ldots, out_n) & = & pabs\ \#\ (y_1, \ldots, y_m) \end{array}$$

formal parameters and the *actual* abstractions transferred will be normalized. This is due to the fact that all process abstractions in the program are normalized and it is impossible to generate 'nameless' process abstractions at runtime.

Notice that under this transformation, the set of *top level* process instantiations monotonically increases. In particular, process instantiations which already were on the top level in the original Eden program still remain on the top level.

2. Forcing normal form evaluation of outputs by a special predefined function `force`: This function is inserted on the right hand side of each equation defining an output contained in the interface of a process in order to force evaluation of the corresponding expression to normal form. Thus, $out = e$, where out denotes an output, is replaced by

$$out = force\ e$$

In Haskell `force` could be defined for special algebraic data types using the predefined function `seq` for sequential composition:

```
force x = nf x 'seq' x
          where
             nf (C x1 ... xn) = force x1 'seq' force x2 'seq'...'seq' force xn
                                         'seq' ()
```

`force` is an identity function which forces the normal form evaluation of its argument via the function `nf`, which traverses its argument completely, thereby forcing evaluation to normal form. Note that `force` is a polytypic function, because the function `nf` must be defined for each algebraic data type. In Haskell it can be defined by a type class for all data types, with an overloaded operation and a corresponding set of instance functions, one for each data type.

In the following, we treat `force` as a predefined function of type `a -> a` and introduce special reduction rules which show the same effect (see Chapter 5.2.2).

Chapter 5

Operational Semantics

The approach. The operational semantics of Eden uses two levels of transition systems. On the upper level *global* effects on process systems are described. The lower level handles the *local* behaviour of processes. Local effects may be connected with global effects on the system level. This is expressed in our semantics via so-called 'actions' (out of a set *Act*). An action associated with a transition of the lower level communicates the need for a global step to the upper level.

Level of abstraction. There are two alternative views of process systems: Firstly, one can assume that complex topologies can be created in one step by creating simultaneously a set of processes which are interconnected in the form of this topology. Secondly, one can assume that the processes are created one by one, and upon process creation connections are built up only between parent and child processes. Process systems created in this way have always the shape of a tree. Other topologies can be established indirectly by use of a so-called bypassing mechanism, which shortcuts connections in order to eliminate intermediate nodes that only copy data from inports to outports. For the description of the language in the previous chapters, we have adopted the first view, whereas for the operational semantics we adopt the second one.

A similar argument applies to the representation of channels. Disregarding implementation issues, the operational semantics models one-by-one transmission for streams and one-message transmission for one-value channels.

5.1 Basic Definitions

The global behaviour of a program is described as a transition relation on the set of system configurations.

$$\models \subset System \times System$$

where a *system configuration* is a set of objects of two kinds: *processes* and *channels*,

$$System = \mathcal{P}(Process \cup Channel).$$

We use the following notation for system configurations: for instance, $\mathcal{S} \oplus \{P, C\}$ denotes a configuration, consisting of a set of objects $\mathcal{S}$ (usually the part of the system that remains unchanged), some process P and some channel C.

Processes are described by their unique process identifier, the input and output interfaces and the so-called environment component:

$$Process = Process\_Id \times \mathcal{P}(Channel\_Descriptor)^2 \times Env_\Sigma.$$

where *Process_Id* is an enumerable set of process identifiers. $Env_\Sigma = Script_\Sigma \cup \{\mathbf{merge}\}$. For user-defined processes, this component contains the script with the local information that the process needs for its internal computation. The nondeterministic predefined process `merge` is specified on the system level only and contains a tag **merge** instead of a script. In this way, a merge process can be distinguished from a user-defined one.

Channels are represented by their global descriptor, the process identifiers of their sender and receiver, and their contents:

$$Channel = Channel\_Descriptor \times Process\_Id^2 \times Contents$$

where *Channel_Descriptor* is a pair:

$$Channel\_Descriptor = Channel\_Id \times Channel\_Tag$$

and *Channel_Id* is an enumerable set of channel identifiers. *Channel_Tag* can take three forms:

- **oneValue**, which is used for one-value channels of arbitrary type
- **stream**, which is used for stream channels, which work like an unbounded buffer. Stream channels are closed by the transfer of the special value *StrmEnd*.
- $type \in IType_\Sigma$ for channel structures. Channel structures denote objects which represent whole data structures of channels which have not yet evolved. In this case, *type* has to be the normalized representation of the type defined in Chapter 4.3.1.

A channel with the channel tag **stream** or **oneValue** will also be called a *regular* channel. The contents of the channels is initially undefined, i.e. $\perp$.

The local behaviour describes the evolution of a single user process and is modelled as a transition relation on the set *Process*, which may also be connected with a global action:

$$\vdash \subset Process \times Act \times Process.$$

Actions are introduced for process creation, in/output, generation of and connection to reply channels, splitting of channel structures and an empty action $\boldsymbol{\tau}$ for internal evaluations without influence on the global system:

$$
\begin{array}{lll}
Act := \{ & (\textbf{create}, p_{child}, I, O, eqs, \tilde{I}, \tilde{O}) & \textbf{process creation} \\
 & (\textbf{send}, v, c), & \textbf{output to channel} \\
 & (\textbf{receive}, v, c), & \textbf{input from channel} \\
 & (\textbf{generate}, d), & \textbf{generation of reply channel} \\
 & (\textbf{connect}, d), & \textbf{connection to reply channel} \\
 & (\textbf{split}, s, t, \tilde{I}) & \textbf{split of channel structure} \\
 & \boldsymbol{\tau}, & \textbf{'empty' action} \\
\mid & v \in Val,\ c \in Channel\_Id,\ p_{child} \in Process\_Id, & \\
 & d, s \in Channel\_Descriptor, \quad t \in T_\Gamma(Channel\_Descriptor), & \\
 & I, O, \tilde{I}, \tilde{O} \subseteq Channel\_Descriptor,\ eqs \in Env_\Sigma \}. &
\end{array}
$$

5.2 Local Evaluations

As user processes are viewed as mappings from input to output channels, the local behaviour of processes is mainly defined by purely functional expressions which cause no effects on the global system. They are accompanied by the empty (or silent) action $\boldsymbol{\tau}$ in the transition relation $\vdash$ of processes, which at the system level has no repercussion, but only changes the internal state of the corresponding process.

In this section we will describe such local computations without any effect on the global network of processes.

5.2.1 Reduction rules for local evaluations

The evaluation of Eden expressions will be described by a reduction relation on *closures*. In general, closures are pairs which consist of an expression and an environment. The latter contains the bindings of the variables and identifiers occurring in the expression. We will represent the environment as a set of equations with normalized right hand sides. The initial environment will be the normalized script with respect to which the expression is going to be evaluated.

Let $\mathcal{A} = \langle A, \alpha \rangle$ be an interpretation (strict continuous algebra) of the basic types and predefined operations, where A is an S-sorted family of complete partial orders and $\alpha : \Omega \to Ops(A)$ maps operation symbols to strict continuous operations over A.
A *closure* is a pair

$$\langle e, eqs \rangle \in Closure(\Sigma, \mathcal{A})$$

where

1. $e \in Exp_\Sigma(V \cup A)$ is a normalized computational expression, i.e. an Eden expression in which values out of the interpretation $\mathcal{A}$ may occur and which is normalized, i.e. only variables occur in argument positions,

2. $eqs \in Script_\Sigma(A, V^{\text{out}})$ with $V \subseteq V^{\text{out}}$, all expressions within eqs are normalized,

and all variable names in *where* expressions, process abstractions etc. in e and eqs are unique.

With these prerequisites we can now define the reduction rules which specify single step expression evaluations within processes:

$$\Rightarrow \subseteq \mathit{Closure}(\Sigma, \mathcal{A}) \times \mathit{Closure}(\Sigma, \mathcal{A})$$

We assume that $A^{bool} = \{tt, ff, \bot^{bool}\}$.

- pattern matching:

  $\langle \texttt{case}\ e\ \texttt{of}\{\ldots; (C_i\ y_{i1} \ldots y_{im}) \to e_i; \ldots\}, eqs\rangle \Rightarrow \langle e_i[y_{i1} \mapsto x_1, \ldots, y_{im} \mapsto x_m], eqs'\rangle$,
  if $\langle e, eqs\rangle \overset{*}{\Rightarrow} \langle (C_i\ x_1 \ldots x_m), eqs'\rangle$

- constant reduction: let $op \in \Omega^{s_1 \to \ldots \to s_n \to s}$, $a_i \in A^{s_i}$

  $\langle (op\ x_1 \ldots x_n), eqs\rangle \Rightarrow \langle \alpha(op)(a_1, \ldots, a_n), eqs^n\rangle$,
  if $\langle x_1, eqs\rangle \overset{*}{\Rightarrow} \langle a_1, eqs^1\rangle, \langle a_2, eqs^1\rangle \overset{*}{\Rightarrow} \langle a_2, eqs^2\rangle \ldots, \langle x_n, eqs^{n-1}\rangle \overset{*}{\Rightarrow} \langle a_n, eqs^n\rangle$

- function rewriting:

  $\langle (f\ x_1 \ldots x_n), eqs \oplus \{f\ y_1 \ldots y_n = e\}\rangle \Rightarrow$
  $\langle e[y_1 \mapsto x_1, \ldots, y_n \mapsto x_n], eqs \oplus \{f\ y_1 \ldots y_n = e\}\rangle$

- process scheme specialization (handling of parameters):

  $$\begin{array}{l}
  \langle\ (p\ x_1 \ldots x_l), \\
  \quad eqs \oplus \{p\ y_1 \ldots y_k = \ \texttt{process}\ (in_1, \ldots, in_m) \to (out_1, \ldots, out_n) \\
  \qquad\qquad\qquad\qquad \texttt{where}\ eqs_p\}\rangle \\
  \Rightarrow \\
  \langle\ p^{\text{new}}, \\
  \quad eqs \oplus\ \{p\ \hat{y}_1 \ldots \hat{y}_k = \ \texttt{process}\ (\widehat{in}_1, \ldots, \widehat{in}_m) \to (\widehat{out}_1, \ldots, \widehat{out}_n) \\
  \qquad\qquad\qquad\qquad \texttt{where} \\
  \qquad\qquad\qquad\qquad eqs_p\ [y_1 \mapsto \hat{y}_1, \ldots, y_k \mapsto \hat{y}_k,\ in_1 \mapsto \widehat{in}_1, \ldots, in_m \mapsto \widehat{in}_m, \\
  \qquad\qquad\qquad\qquad\quad out_1 \mapsto \widehat{out}_1, \ldots, out_n \mapsto \widehat{out}_n], \\
  \qquad\qquad p^{\text{new}} y_{l+1} \ldots y_k = \ \texttt{process}\ (in_1, \ldots, in_m) \to (out_1, \ldots, out_n) \\
  \qquad\qquad\qquad\qquad \texttt{where} \\
  \qquad\qquad\qquad\qquad eqs_p\ [y_1 \mapsto x_1, \ldots, y_l \mapsto x_l] \oplus \bigoplus_{i=1}^{l} \mathit{closure}(x_i, eqs)\}\rangle,
  \end{array}$$

  where $\hat{\cdot}$ is used to introduce fresh variable names.

  The function $\mathit{closure}(y, eqs)$ collects all equations within eqs on which the definition of y depends:

  $$\begin{array}{lcl}
  \mathit{closure}(y, eqs \oplus \{y = e\}) & = & \{y = e\} \cup \bigcup_{z \in \mathit{vars}(e)} \mathit{closure}(z, eqs) \\
  \mathit{closure}(y, eqs) & = & \emptyset, \text{ if } y \text{ occurs in no left hand side in } eqs.
  \end{array}$$

  Note that new equations are introduced for the process parameters, because they will be evaluated by the new process[1]. At this point work may be duplicated, as unevaluated expressions are copied into the state of the newly created process. This

[1] Note that apart from the parameters $x_1 \ldots x_l$, no free variables have to be handled, because Kernel Eden requires process abstractions to be *closed* objects.

convention is chosen in order to remove work from the parent process and thereby foster the spreading of parallelism in a distributed setting.

The variables within the original process abstraction are renamed in order to ensure unique variable names within scripts.

- let reduction: $\langle$ `let` loc `in` e, eqs $\rangle \Rightarrow \langle e, eqs \oplus loc\rangle$.

  Note that no renaming is necessary, because we presuppose that bound and free variables within scripts are unique.

- variable evaluation and update:

  $\langle x, eqs \oplus \{x = e\}\rangle \Rightarrow \langle e^{\text{hnf}}, eqs' \oplus \{x = e^{\text{hnf}}\}\rangle$,

  if $\langle e, eqs\rangle \overset{*}{\Rightarrow} \langle e^{\text{hnf}}, eqs'\rangle$ and e^{hnf} is in head normal form, i.e. a value, a constructor application, a partial application, a process abstraction, or an expression defining the generation of or the connection to a reply channel.

5.2.2 Reduction rules for `force`

In order to force normal form evaluation, we have introduced the special predefined function `force` in the equations defining outports. For the evaluation of this function special reduction rules are provided:

- for each expression e:
  $\langle$`force` $e, eqs\rangle \Rightarrow \langle$`force` $e', eqs'\rangle$, if $\langle e, eqs\rangle \Rightarrow \langle e', eqs'\rangle$

- for each n-ary algebraic data constructor C $(n > 1)$:
  $\langle$`force` $(C\ e_1 \ldots e_n), eqs\rangle \Rightarrow \langle (C$ (`force` $e_1) \ldots$ (`force` $e_n)), eqs\rangle$

- for each partial application of a function or constructor φ:
  $\langle$`force` $(\varphi\ e_1 \ldots e_n), eqs\rangle \Rightarrow \langle (\varphi$ (`force` $e_1) \ldots$ (`force` $e_n)), eqs\rangle$

- for each nullary constructor or constant C:
  $\langle$`force` $C, eqs\rangle \Rightarrow \langle C, eqs\rangle$

- for each value $a \in A$:
  $\langle$`force` $a, eqs\rangle \Rightarrow \langle a, eqs\rangle$

- variables:
  $\langle$`force` $x, eqs \oplus \{x = e\}\rangle \Rightarrow \langle x, eqs \oplus \{x =$ `force` $e\}\rangle$

- process instantiations:
  $\langle$`force` $o_i,\quad eqs \oplus \{(o_1, \ldots, o_i, \ldots, o_n) = pn$ `#` $(i_1, \ldots, i_m),\ pn = e\}\rangle \Rightarrow$
  $\langle o_i,\quad eqs \oplus \{(o_1, \ldots, o_i, \ldots, o_n) = pn$ `#` $(i_1, \ldots, i_m),\ pn =$ `force` $e\}\rangle$

5.2.3 Local evaluations in the context of a process and the system

The action $\boldsymbol{\tau}$ is used to model process transitions which have no effect on the global system. There is always demand for the evaluation of outports.

$$\boxed{\begin{array}{l}\langle p, In, Out \oplus \{o\}, eqs\rangle \vdash_\tau \langle p, In, Out \oplus \{o\}, eqs'\rangle \\ \text{if } \langle o, eqs\rangle \overset{*}{\Rightarrow} \langle e, eqs'\rangle\end{array}}$$

In this case, the information $\{o = e\}$ is contained in eqs'.

Such an internal computation of a process $P \vdash_\tau P'$ does not have any global effect on the system:

$$\boxed{\mathcal{S} \oplus \{P\} \models \mathcal{S} \oplus \{P'\}, \quad \text{if } P \vdash_\tau P'}$$

5.3 Process Creation

If the internal evaluation of a process gives rise to the instantiation of child processes[2], not only the global system is changed by the generation of these new children, but also the internal state of the parent is changed by the incorporation of new in/outports communicating with them.

For simplicity we will first describe a process instantiation which builds up connections between the parent and the child process only. Later we will show how channels can be bypassed immediately on process creation to establish direct connections between producer and consumer processes.

When a process activates a child process, it generates an action

$$a = (\mathbf{create}, p_{child}, I, O, eqn, \tilde{I}, \tilde{O})$$

where

- p_{child} is the identification of the new process,
- I and O are sets with newly created descriptors for the sets of inputs and outputs of the new process (see Chapter 4.3.1), where m and n denote the numbers of input channels and output channels, respectively, and
- eqn is the set of equations (script) with the local definitions for a new user-defined process, or the tag **merge** for a merge process, and
- $\tilde{I}, \tilde{O}$ are sets with descriptors of existing channels which will be bypassed to the new process. For the moment we ignore $\tilde{I}$ and $\tilde{O}$. They will be taken into account in Chapter 5.3.2.

A process abstraction uses local names for its in/outputs but, on process instantiation, communication channels will be established with global identifiers for the use of the system. Therefore, the local names of the ports will be replaced by the corresponding channel identifiers which are treated as variables. A global side-effecting procedure *newNr* provides previously unused identifiers. Together with a function *tag* (which takes as its argument the annotated type of the in- or output), a new descriptor $(id, tag(type))$ for a local input or output can be generated:

[2]There will always be demand for the evaluation of executable process instantiations within scripts.

$$tag(t_c) = \begin{cases} \textbf{stream} & \text{if } t_c \in MType_\Sigma, t_c = <[a]> \text{ for some } a \in MType_\Sigma \\ \textbf{oneValue} & \text{if } t_c \in MType_\Sigma, t_c \neq <[a]> \text{ for all } a \in MType_\Sigma \\ t_c & \text{if } t_c \text{ where } t_c \in IType_\Sigma \setminus MType_\Sigma \end{cases}$$

In this case, t_c is a fully instantiated and annotated type, which is available, because now, on process creation, polymorphism is resolved. Note that it not sufficient to mark channel structures with a special "tag", because there can be nested structures of channels (see Chapter 5.6.1). In the operational semantics, we will therefore store the full type with annotations.

5.3.1 Basic transitions without bypassing

In the following, we use a parameterless process abstraction pn, the body of which does not depend on any global variable[3].

The component types in the normal form of the type[4] pt of the process abstraction pn are instantiated by a suitable type substitution ϑ that depends on the expressions found in the instantiation. Substituting the types ip_j and op_j yields it_j and ot_j, respectively. These types are needed in order to assign the correct tags to the new input and output channels. The set O containing descriptors of new channels *from* the child process is added to the parent's inputs. Correspondingly, the set I of descriptors of new channels *to* the child process is added to the parent's outputs.

The system is extended with the newly created process and with new channels and channel structures. The IC_k are the full representations of the new channel descriptors contained in I, and OC_j those in O, respectively.

[3]This may happen if a process parameter expression depends on an input variable of the parent process. If this dependency involved strictness in this variable, the child process could only be created after this input had been *transmitted in full.* This pathological case should be avoided by transforming such a complex parameter into an input.

[4]Here we need the annotated types, as generated by the function *ptype_nf* (see Chapter 4.3.1). In the following we will generalize the Haskell notation :: to mean *has annotated type.*

$$
\begin{array}{lll}
\langle p_{parent}, & In, & Out, \quad eqs_{parent} \oplus \\
 & & \{\, pn = \texttt{process}\ (s_1, \ldots, s_m)\ \texttt{->}\ (r_1, \ldots, r_n) \\
 & & \qquad \texttt{where}\ eqs_{pn}, \\
 & & \quad (o_1, \ldots, o_n) = pn\ \texttt{\#}\ (i_1, \ldots, i_m)\} \rangle \\
 & & \text{with } pn :: pt, \\
 & & ptype\_nf\,(pt) = \mathbf{Process}\,(ip_1, \ldots, ip_m)\,(op_1, \ldots, op_n) \\
 & & \left.\begin{array}{l} i_j :: it_j = ip_j\vartheta \text{ for } j = 1, \ldots, m \\ o_j :: ot_j = op_j\vartheta \text{ for } j = 1, \ldots, n \end{array}\right\} \vartheta \text{ suitable type subst.}
\end{array}
$$

$$
\vdash_{(\mathbf{create}, p_{child}, I, O, eqs'_{pn}, \emptyset, \emptyset)}
$$

$$
\begin{array}{lll}
\langle p_{parent}, & In \cup O, & Out \cup I, \quad eqs'_{parent} \oplus \\
 & & \{\, pn = \texttt{process}\ (s_1, \ldots, s_m)\ \texttt{->}\ (r_1, \ldots, r_n) \\
 & & \qquad \texttt{where}\ eqs_{pn}\}\rangle
\end{array}
$$

where
$I = \{\, (newNr(i_1), tag(it_1)), \ \ldots, (newNr(i_m), tag(it_m))\, \}$ and
$O = \{\, (newNr(o_1), tag(ot_1)), \ldots, (newNr(o_n), tag(ot_n))\, \}$
contain descriptors for the newly created channels,

$$
\begin{array}{rl}
eqs'_{parent} = & eqs_{parent}[o_j \mapsto newNr(o_j) \mid 1 \leq j \leq n] \oplus \\
 & \{newNr(i_k) = \texttt{force}\ i_k \mid 1 \leq k \leq m\} \\
eqs'_{pn} = & eqs_{pn}[s_1 \mapsto newNr(i_1), \ldots, s_m \mapsto newNr(i_m), \\
 & r_1 \mapsto newNr(o_1), \ldots, r_n \mapsto newNr(o_n)].
\end{array}
$$

$$
\begin{array}{ll}
 & \mathcal{S}\oplus \ \{P_{parent}\} \\
\models & \mathcal{S}\oplus \ \{P'_{parent}, P_{child}\} \oplus \bigoplus_{k=1}^{n}\{IC_k\} \oplus \bigoplus_{j=1}^{m}\{OC_j\}
\end{array}
$$

if $P_{parent} = \langle p_{parent}, In, Out, eqs_{parent}\rangle \ \vdash_{(\mathbf{create}, p_{child}, I, O, eqs'_{pn}, \emptyset, \emptyset)} \ P'_{parent}$

$$
\begin{array}{llll}
\text{where} & P_{child} & = & \langle p_{child}, I, O, eqs'_{pn}\rangle, \\
 & IC_k & = & \langle ic_k, (p, p_{child}), \bot\rangle \quad \text{for each } k \in \{1, \ldots, n\}, \quad ic_k \in I \\
 & OC_j & = & \langle oc_j, (p_{child}, p), \bot\rangle \quad \text{for each } j \in \{1, \ldots, m\}, \quad oc_j \in O
\end{array}
$$

The rule for instantiating the non-deterministic predefined process `merge` is given in Chapter 5.7.

5.3.2 The integration of bypassing

The above rule for process instantiation leads to the creation of process trees, which reflect the generation history of the process system. Thus, direct communication is restricted to occur between parents and children, and communication between arbitrary nodes within the process system has to go via common ancestor processes. In order to eliminate such 'detours' and establish direct connections between producer and consumer processes, process creations have to be analyzed in more detail. In the following, we will give an example[5] of automatic bypassing.

[5]This example serves as an illustration of the mechanism and as a bridge between the two alternative views of topologies discussed at the beginning of this chapter.

Example 5.1

Reconsider the 2×2 grid defined in Example 2.1: We expect the connections within the grid, e.g. `right11` and `bot11`, to be modelled by direct channels between the corresponding processes, e.g. from process in position 11 to processes 12 and 21, respectively. But the simple process instantiation rule given above generates two channels for each connection: one from process 11 to the grid process (i.e. the parent process) and another from the grid process to the processes 12 or 21 respectively. For the `right11` connection the grid process copies the values arriving from process 11 to the channel leading to process 12.

This useless copying and the double transfer can easily be avoided. If we notice on process creation that a channel to or from the new process is simply connected to an already existent channel and if the data transmitted is not used elsewhere within the parent process, we can suppress the generation of a new channel and simply redirect the existing channel. This is the case if the identifier of an already existing channel does only occur within the process instantiation and not in the remainder of the process[6]. Consider for example the following situation in the evaluation of the process grid, where $\langle p, In, Out, Eqs\rangle$ denotes the process:

$$
\begin{array}{rll}
In = & \{ & \overline{(id_{\texttt{left11}}, t_{\texttt{left11}})},\ (id_{\texttt{left21}}, t_{\texttt{left21}}),\ \overline{(id_{\texttt{top11}}, t_{\texttt{top11}})},\ (id_{\texttt{top12}}, t_{\texttt{top12}})\ \}, \\
Out = & \{ & (id_{\texttt{right12}}, t_{\texttt{right12}}),\ (id_{\texttt{right22}}, t_{\texttt{right22}}),\ (id_{\texttt{bot21}}, t_{\texttt{bot21}}),\ (id_{\texttt{bot22}}, t_{\texttt{bot22}})\ \}, \\
Eqs = & \{ & \underline{(\texttt{right11, bot11})}\ \ \texttt{= pabs \# (}\ \overline{id_{\texttt{left11}}, id_{\texttt{top11}}}\ \texttt{), pabs = process} \ldots \} \oplus \\
 & \{ & (id_{\texttt{right12}}\texttt{,bot12})\ \ \texttt{= pabs \# (right11,}id_{\texttt{top12}}\texttt{)}, \\
 & & \texttt{(right21,}id_{\texttt{bot21}}\texttt{)}\ \ \texttt{= pabs \# (}id_{\texttt{left21}}\texttt{,bot11)}, \\
 & & (id_{\texttt{right22}}\texttt{,}id_{\texttt{bot22}})\ \ \texttt{= pabs \# (right21,bot12)}\ \}
\end{array}
$$

The channel identifiers $id_{\texttt{left11}}$ and $id_{\texttt{top11}}$ occur only within the first process instantiation. Therefore it is not necessary to create new channels, but it is sufficient to redirect the existing channels to the process in position 11 (marked by overlines). New channels will only be created for the output channels of process 11 (marked by underlines). Note that the annotated type of all horizontal channels is `a` and that of all vertical channels is `b` and consequently for the tags of the new channels the following equalities hold: $t_{\texttt{right11}} = t_{\texttt{left11}}$ and $t_{\texttt{bot11}} = t_{\texttt{top11}}$. Thus after the creation of process 11, the grid process will have the following configuration:

$$
\begin{array}{rll}
In = & \{ & (id_{\texttt{left21}}, t_{\texttt{left21}}),\ (id_{\texttt{top12}}, t_{\texttt{top12}}),\ \underline{(id_{\texttt{right11}}, t_{\texttt{right11}}),\ (id_{\texttt{bot11}}, t_{\texttt{bot11}})}\ \}, \\
Out = & \{ & (id_{\texttt{right12}}, t_{\texttt{right12}}),\ (id_{\texttt{right22}}, t_{\texttt{right22}}),\ (id_{\texttt{bot21}}, t_{\texttt{bot21}}),\ (id_{\texttt{bot22}}, t_{\texttt{bot22}})\ \}, \\
Eqs = & \{ & \texttt{pabs = process ...}\} \oplus \\
 & \{ & (id_{\texttt{right12}}\texttt{,bot12})\ \ \texttt{= pabs \# (right11,}id_{\texttt{top12}}\texttt{)}, \\
 & & \texttt{(right21,}id_{\texttt{bot21}}\texttt{)}\ \ \texttt{= pabs \# (}id_{\texttt{left21}}\texttt{,bot11)}, \\
 & & (id_{\texttt{right22}}\texttt{,}id_{\texttt{bot22}})\ \ \texttt{= pabs \# (right21,bot12)}\ \}
\end{array}
$$

[6]Due to lazy evaluation, this condition is of course sufficient, but not necessary. A usage analysis based on abstract interpretation would provide a more precise criterion. However, our intention here is to keep the tests necessary for bypassing as cheap as possible.

Such an optimization step will be carried out for each of the four child processes of process `grid`, always redirecting one of its current inports and outports to the new child process, finally resulting in the topology shown in the diagram in Example 2.1. ◁

The general rules for process instantiation are given below. The arguments $\tilde{I}$ and $\tilde{O}$ of the generated **create**-action represent existing in/output channel identifiers which will be bypassed to the new process. Bypassing a channel means diverting an existing channel to or from the parent process instead of creating a new channel. This means, the child replaces the parent as the sender or receiver of this channel. Likewise, the channels in $\tilde{I}$ and $\tilde{O}$ are removed from the parent's interface.

In the scheme for the system, the ic_i and oc_k denote the whole *descriptors* of the channels, i.e. (id, tag). As $\tilde{I}$ and $\tilde{O}$ contain the descriptors of the bypassed channels, which are not affected by bypassing, no $\tilde{I}'$ and $\tilde{O}'$ have to be introduced.

$$
\begin{array}{lll}
\langle p_{parent},\ In, & Out, & eqs_{parent} \oplus \\
 & & \{\ pn = \texttt{process}\ (s_1, \ldots, s_m)\ \texttt{->}\ (r_1, \ldots, r_n) \\
 & & \qquad \texttt{where}\ eqs_{pn}, \\
 & & \quad (o_1, \ldots, o_n) = pn\ \texttt{\#}\ (i_1, \ldots, i_m)\}\ \rangle \\
 & & \text{with } pn :: pt, \\
 & & ptype\_nf(pt) = \mathbf{Process}\,(ip_1, \ldots, ip_m)\,(op_1, \ldots, op_n) \\
 & & \left.\begin{array}{l} i_j :: it_j = ip_j\vartheta \text{ for } j = 1, ..., m \\ o_j :: ot_j = op_j\vartheta \text{ for } j = 1, ..., n \end{array}\right\} \vartheta \text{ suit. type subst.}
\end{array}
$$

$$
\vdash_{(\mathbf{create},\ p_{child},\ I,\ O,\ eqs'_{pn},\ \tilde{I},\ \tilde{O})}
$$

$$
\begin{array}{lll}
\langle p_{parent},\ In \cup O \setminus \tilde{I}, & Out \cup I \setminus \tilde{O}, & eqs'_{parent} \oplus \\
 & & \{\ pn = \texttt{process}\ (s_1, \ldots, s_m)\ \texttt{->}\ (r_1, \ldots, r_n) \\
 & & \qquad \texttt{where}\ eqs_{pn}\}\rangle
\end{array}
$$

$$
\begin{array}{lll}
\text{where} & I & = \{(s'_k, tag(it_k)) \mid i_k \in vars(eqs_{parent}), 1 \le k \le m\}, \\
 & O & = \{(r'_k, tag(ot_k)) \mid o_k \in vars(eqs_{parent}), 1 \le k \le n\} \\
\text{and} & \tilde{I} & = \{(i_j, tg_j) \in I \mid i_j \notin vars(eqs_{parent}), 1 \le j \le m\}, \\
 & \tilde{O} & = \{(o_j, tg_j) \in O \mid o_j \notin vars(eqs_{parent}), 1 \le j \le n\}^7
\end{array}
$$

$$
\text{with } s'_k = \begin{cases} i_k & \text{if } (i_k, tg_k) \in \tilde{I} \\ newNr(i_k) & \text{otherwise} \end{cases} \quad \text{and} \quad r'_k = \begin{cases} o_k & \text{if } (o_k, tg_k) \in \tilde{O} \\ newNr(o_k) & \text{otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
eqs'_{parent} & = & eqs_{parent}[\{o_j \mapsto r'_j \mid o_j \in vars(eqs_{parent}), 1 \le j \le n\}] \\
 & & \oplus\ \{s'_k = \texttt{force}\ i_k \mid i_k \in vars(eqs_{parent}), 1 \le k \le m\}, \quad \text{and} \\
eqs'_{pn} & = & eqs_{pn}[s_1 \mapsto s'_1, \ldots, s_m \mapsto s'_m, r_1 \mapsto r'_1, \ldots, r_n \mapsto r'_n]
\end{array}
$$

[7] As already mentioned in the previous example, the criterion for bypassing is the simplest possible: channels the names of which are not found in the remainder of the body are bypassed.

$$
\begin{array}{rlll}
 & \mathcal{S} \oplus & \{P_{parent}\} & \oplus \bigoplus_{k=1}^{r}\{\widetilde{IC}_k\} \oplus \bigoplus_{j=1}^{s}\{\widetilde{OC}_j\} \\
\models & \mathcal{S} \oplus & \{P'_{parent}, P_{child}\} & \oplus \bigoplus_{k=1}^{r}\{\widetilde{IC'}_k\} \oplus \bigoplus_{j=1}^{s}\{\widetilde{OC'}_j\} \\
 & & & \oplus \bigoplus_{k=1}^{n}\{IC_k\} \oplus \bigoplus_{k=1}^{m}\{OC_j\}
\end{array}
$$

$$
\begin{array}{llll}
\text{where} & P_{child} & = & \langle p_{child}, I \cup \tilde{I}, O \cup \tilde{O}, eqn\rangle \\
 & \widetilde{IC'}_k & = & \langle \widetilde{ic}_k, (p_k, p_{child}), ivals_k\rangle \quad \text{for each } k \in \{1,\ldots,r\} \\
 & \widetilde{OC'}_j & = & \langle \widetilde{oc}_j, (p_{child}, p_j), ovals_j\rangle \quad \text{for each } j \in \{1,\ldots,s\} \\
 & IC_k & = & \langle ic_k, (p, p_{child}), \bot\rangle \quad \text{for each } k \in \{1,\ldots,n\} \\
 & OC_j & = & \langle oc_j, (p_{child}, p), \bot\rangle \quad \text{for each } j \in \{1,\ldots,m\} \\
 & I & = & \{ic_k \mid 1 \le k \le n\} \\
 & O & = & \{oc_j \mid 1 \le j \le m\} \\
 & \tilde{I} & = & \{\widetilde{ic}_k \mid 1 \le k \le r\} \\
 & \tilde{O} & = & \{\widetilde{oc}_j \mid 1 \le j \le s\} \\
 & \widetilde{IC}_k & = & \langle \widetilde{ic}_k, (p_k, p), ivals_k\rangle \quad \text{for each } k \in \{1,\ldots,r\}^8 \\
 & \widetilde{OC}_j & = & \langle \widetilde{oc}_j, (p, p_j), ovals_j\rangle \quad \text{for each } j \in \{1,\ldots,s\}
\end{array}
$$

$$P_{parent} = \langle p_{parent}, In, Out, eqs_{parent}\rangle \vdash_{(\mathbf{create}, p_{child}, I, O, eqn, \tilde{I}, \tilde{O})} P'_{parent}$$

[8]Note that bypassed channels are not necessarily empty. In the case that the intended receiver is created after the sender, the channel can already contain data before the final bypassed connection is established.

5.4 Communication

5.4.1 Sending values

Writing a value to an output channel is reflected by a **send**-action, specifying the transmitted value and the channel identifier. The value that is transmitted will be a fully evaluated first-order term, a function or a process abstraction, but never a suspension.

Senders do not close any channels, but simply eliminate the outport from their interface after the transmission of the final value. The closing (elimination) of channels is performed by the receiver after it has completely consumed the contents of the channel buffer (see Chapter 5.4.2).

On the system level, sending a value means writing it to (the write end of) the channel buffer, if it is not *StrmEnd*. Figure 5.1 shows the influence of send- and receive operations on a stream channel. The diagram illustrates that a channel is a 1:1 connection between two processes with a buffer.

Sending a value to a stream channel. If the sender has evaluated some value v_{k+1} to normal form, it produces a **send** action. This action informs the system that this value has to be inserted at the write end of the channel.

$$
\begin{array}{rlll}
 & \langle p_{send}, & In, & Out \cup \{(c_{out}, \mathbf{stream})\}, \quad eqs \oplus \{c_{out} = v_{k+1} : rest\}\rangle \\
\vdash_{(\mathbf{send}, v, c_{out})} & \langle p_{send}, & In, & Out \cup \{(c_{out}, \mathbf{stream})\}, \quad eqs \oplus \{c_{out} = rest\}\rangle
\end{array}
$$

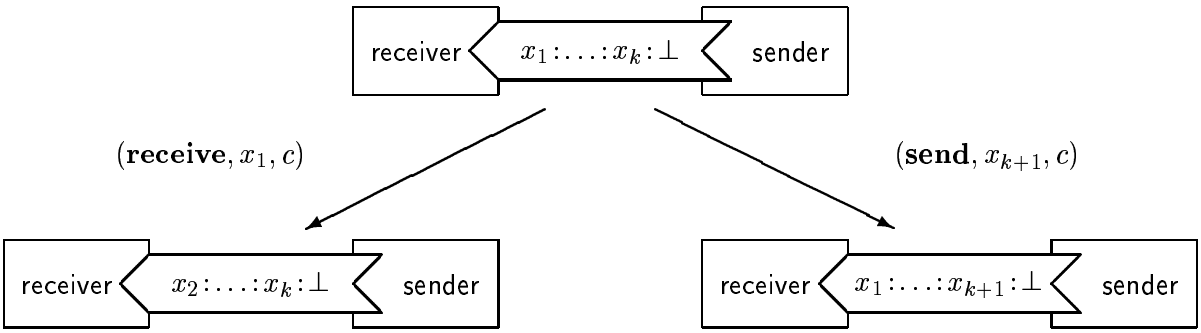


Figure 5.1: Stream communication: the effect of a **receive** action on stream c is shown on the left, that of a **send** action on the right.

$$\begin{array}{rl} & \mathcal{S} \oplus \{P_{send}, \langle (c_{out}, \mathbf{stream}), (p_{send}, p_{rec}), v_1 : v_2 : \ldots : v_k : \bot \rangle\} \\ \models & \mathcal{S} \oplus \{P'_{send}, \langle (c_{out}, \mathbf{stream}), (p_{send}, p_{rec}), v_1 : v_2 : \ldots : v_k : v_{k+1} : \bot \rangle\} \\ \text{if } P_{send} \vdash_{(\mathbf{send}, v_{k+1}, c_{out})} P'_{send} & \end{array}$$

Sending *strmEnd* to a stream channel. If the sender reaches the end of a stream output, it sends the special closing value *StrmEnd* to the channel and eliminates the outport. On the system level, the empty list is written to the previously undefined end of the channel buffer.

$$\begin{array}{rlll} & \langle p_{send}, & In, & Out \oplus \{(c_{out}, \mathbf{stream})\}, & eqs \oplus \{c_{out} = [\,]\} \rangle \\ \vdash_{(\mathbf{send}, StrmEnd, c_{out})} & \langle p_{send}, & In, & Out, & eqs \rangle \end{array}$$

$$\begin{array}{rl} & \mathcal{S} \oplus \{P_{send}, \langle (c_{out}, \mathbf{stream}), (p_{send}, p_{rec}), b_1 : b_2 : \ldots : b_k : \bot \rangle\} \\ \models & \mathcal{S} \oplus \{P'_{send}, \langle (c_{out}, \mathbf{stream}), (p_{send}, p_{rec}), b_1 : b_2 : \ldots : b_k : [\,] \rangle\} \\ \text{if } P_{send} \vdash_{(\mathbf{send}, strmEnd, c_{out})} P'_{send} & \end{array}$$

Sending "the" value to a one-value channel. If the sender has computed the normal form of the value v to be sent to a one-value channel, it can perform a **send** action and eliminate the outport. The system then overwrites the $\bot$ in the channel buffer with this value.

$$\begin{array}{rlll} & \langle p_{send}, & In, & Out \cup \{(c_{out}, \mathbf{oneVal})\}, & eqs \oplus \{c_{out} = v\} \rangle \\ \vdash_{(\mathbf{send}, v, c_{out})} & \langle p_{send}, & In, & Out, & eqs \rangle \end{array}$$

$$\begin{array}{rl} & \mathcal{S} \oplus \{P_{send}, \langle (c_{out}, \mathbf{oneVal}), (p_{send}, p_{rec}), \bot \rangle\} \\ \models & \mathcal{S} \oplus \{P'_{send}, \langle (c_{out}, \mathbf{oneVal}), (p_{send}, p_{rec}), v \rangle\} \\ \text{if } P_{send} \vdash_{(\mathbf{send}, v, c_{out})} P'_{send} & \end{array}$$

5.4.2 Receiving values

Input operations are reflected by **receive** actions. On the system level, a **receive** action triggers the transfer of a value from the channel buffer into the environment of the process. The rest can also be undefined, of course. An attempt to read this undefined rest will block, i.e. no **receive** action will be possible at this point of time.

Receiving a value from a stream channel.

$$\begin{array}{rll} & \langle p_{rec},\ In \oplus \{(c_{in}, \textbf{stream})\},\ & Out,\ \ eqs\rangle \\ \vdash_{(\textbf{receive},v,c_{in})} & \langle p_{rec},\ In \oplus \{(c_{in}, \textbf{stream})\},\ & Out,\ \ eqs[c_{in} \mapsto (v : c_{in})]\rangle \end{array}$$

$$\begin{array}{rl} & \mathcal{S} \oplus \ \{P_{rec},\ \ \langle (c_{in}, \textbf{stream}), (p_{send}, p_{rec}), (v : rest)\rangle\} \\ \models & \mathcal{S} \oplus \ \{P'_{rec},\ \ \langle (c_{in}, \textbf{stream}), (p_{send}, p_{rec}), rest\rangle\} \\ & \text{if } P_{rec} \vdash_{(\textbf{receive},v,c_{in})} P'_{rec} \end{array}$$

Receiving *strmEnd* from a stream channel.

$$\begin{array}{rll} & \langle p_{rec},\ In \oplus \{(c_{in}, \textbf{stream})\},\ & Out,\ \ eqs\rangle \\ \vdash_{(\textbf{receive},strmEnd,c_{in})} & \langle p_{rec},\ In,\ & Out,\ \ eqs[c_{in} \mapsto [\,]]\rangle \end{array}$$

$$\begin{array}{rl} & \mathcal{S} \oplus \ \{P_{rec},\ \ \langle (c_{in}, \textbf{stream}), (p_{send}, p_{rec}),\ [\,]\rangle\} \\ \models & \mathcal{S} \oplus \ \{P'_{rec}\} \\ & \text{if } P_{rec} \vdash_{(\textbf{receive},strmEnd,c_{in})} P'_{rec} \end{array}$$

Receiving "the" value from a one-value channel.

$$\begin{array}{rll} & \langle p_{rec},\ In \oplus \{(c_{in}, \textbf{oneVal})\},\ & Out,\ \ eqs\rangle \\ \vdash_{(\textbf{receive},v,c_{in})} & \langle p_{rec},\ In,\ & Out,\ \ eqs[c_{in} \mapsto v]\rangle \end{array}$$

$$\begin{array}{rl} & \mathcal{S} \oplus \ \{P_{rec},\ \ \langle (c_{in}, \textbf{oneVal}), (p_{send}, p_{rec}),\ v\rangle\} \\ \models & \mathcal{S} \oplus \ \{P'_{rec}\} \\ & \text{if } P_{rec} \vdash_{(\textbf{receive},v,c_{in})} P'_{rec} \end{array}$$

The handling of receive actions for *channel structures* is analogous to the above cases and will be explained in Chapter 5.6.2.

5.5 Dynamic Reply Channels

5.5.1 Channel generation

Processes are able to produce a reply channel and send a reference to it to another process. This is done by a special action **generate**, providing a new channel identifier c_{new}. This identifier is used to replace the channel variable `cCont` in the process environment. Furthermore, the pair (dyn,c_{new}) is a value of type *Chan_name* `a` that substitutes for `cName` and can be written to some output, i.e. used as a reference to `cCont`. The value of the **tag** depends on `a` and can be either **oneVal** or **stream**[9]. This will be determined by type inference in the body of the process abstractions of the processes involved in this communication. If this is not possible due to separate compilation and unresolved polymorphism, the tag can also be instantiated with $\perp$ and resolved on the **connect**-action (see Chapter 5.5.2).

On the system level, the **generate** action induces the creation of a new channel whose sender is still unknown.

[9]Reply channels always represent regular channels, not channel structures.

$$\begin{array}{lll} \langle p_{gen}, In, & Out, & eqs \oplus \{y = \texttt{force}\ (\texttt{new}\ (\texttt{cName}, \texttt{cCont})\ e)\rangle \\ \vdash_{(\textbf{generate},\ (c_{new},\ tg))} & & \\ \langle p_{gen}, In \oplus \{(c_{new}, tg)\}, & Out, & eqs \oplus \{y = \\ & & \texttt{force}\ e[\texttt{cName} \mapsto (\text{dyn}, c_{new}), \texttt{cCont} \mapsto c_{new}]\}\rangle \end{array}$$

where $c_{new} \in Channel\_Id$ is a new channel identifier.

$$\begin{array}{rl} & \mathcal{S} \oplus \{P_{gen}\} \\ \models & \mathcal{S} \oplus \{P'_{gen}, \quad \langle (c_{in}, tg), (\bot, p_{gen}), \bot\rangle\} \\ & \text{if } P_{gen} \vdash_{(\textbf{generate},\ (c_{in},\ tg))} P'_{gen} \end{array}$$

5.5.2 Connecting to a dynamic channel

When a process has received a dynamic reply channel and decides to use it for sending information, it has to extend its interface correspondingly. This is done by a **connect** action, which adds the channel identifier to the set of outports and its defining equation to the environment.

On the system level, a **connect** action replaces the $\bot$ in the position of the sender by the process identifier of the corresponding process. Note that this transition can only be carried out if no other process has connected to the channel before (*receive and use*).

$$\begin{array}{lll} \langle p_{send}, In, & Out, & eqs \oplus \{y = \texttt{force}\ (\text{dyn}, c_{out})\ \texttt{!*}\ e_1\ \texttt{par}\ e_2\rangle \\ \vdash_{(\textbf{connect},\ (c_{out},\ tg))} & & \\ \langle p_{send}, In, & Out \oplus \{(c_{out}, tg)\}, & eqs \oplus \{y = \texttt{force}\ e_2, \quad c_{out} = \texttt{force}\ e_1\}\rangle \end{array}$$

$$\begin{array}{rl} & \mathcal{S} \oplus \{P_{send}, \quad \langle (c_{out}, \textbf{tag}), \quad (\bot, p_{rec}), \bot\rangle\} \\ \models & \mathcal{S} \oplus \{P'_{send}, \quad \langle (c_{out}, \textbf{tag}), \quad (p_{send}, p_{rec}), \bot\rangle\} \\ & \text{if } P_{send} \vdash_{(\textbf{connect},\ (c_{out},\ tg))} P'_{send} \end{array}$$

5.6 Channel Structures

This section describes the dynamic handling of data structures of communication channels (cf. Chapter 5.4). For explanatory purposes, we will start this section with a detailed discussion of channel lists in the absence of bypassing. This will also be the most frequent case in practice. In the subsequent subsections we will then introduce the handling of general data structures and bypassing. Note that polymorphism doesn't have to be handled because the steps described here are carried out after process creation.

5.6.1 Sending to channel structures

Splitting of channel lists without bypassing

If we see in the interface that an identifier represents a channel list and we encounter an equation defining this list, the step we have to take depends on the constructor used on the right hand side of this definition:

o^{list} = **Cons h t** means that we have to generate a new (regular) channel for the head **h** and a new channel list for **t**. This is expressed in the operational semantics by means of an action (**split**, *structChan*, `Cons` o^i *structChan'*, $\emptyset$) that informs the system about the splitting up of *structChan* into a channel with descriptor o^i and some channel list *structChan'*. The empty set represents the set of channel identifiers that can be bypassed, which will be discussed in Chapter 5.6.1. The system will generate the new channels and insert the term into the contents of the channel structure that is split up.
Let the identifier o^{list} represent a channel list of type `o`:

$$\langle (o^{list}, [< \mathtt{o} >]), (p_s, p_r), cont \rangle$$

Note that there is no need to introduce `force` manually, because o^{list} has been forced initially and this has propagated to the components already.

On the system level, the split action modifies the respective channel structure. The corresponding changes in the receiver process are introduced on receive, which is advisable both in order to model the asynchronous model of communication and in order to alleviate the lazy access to inputs.

$$\begin{array}{ll} \langle p_s,\ In,\ \ Out \oplus \{(o^{list}, [< \mathtt{o} >])\}, & eqs \oplus \{o^{list} = x : xs\} \rangle \\ \vdash_{(\mathbf{split}, (o^{list}, [<\mathtt{o}>]), \mathtt{Cons}\ (o^i, tag(<o>)\ (o^{rest}, [<\mathtt{o}>]), \emptyset)} & \\ \langle p_s,\ In,\ \ Out \oplus \{(o^i, tag(< o >)), (o^{rest}, [< \mathtt{o} >])\}, & eqs \oplus \{o^i = x\} \oplus \{o^{rest} = xs\} \rangle, \end{array}$$

$$\begin{array}{lll} & \mathcal{S} \oplus & \{P_{send},\ \ \langle (o^{list}, [< \mathtt{o} >]), \quad (p_s, p_r), \bot \rangle\} \\ \models & \mathcal{S} \oplus & \{P'_{send},\ \ \langle (o^{list}, [< \mathtt{o} >]), \quad (p_s, p_r), Cons\ (o^i, tag(< o >)\ (o^{rest}, [< \mathtt{o} >]) \rangle, \\ & & \qquad\qquad \langle (o^i, tag(< o >)), \quad (p_s, p_r), \bot \rangle, \\ & & \qquad\qquad \langle (o^{rest}, [< \mathtt{o} >]), \quad (p_s, p_r), \bot \rangle\} \\ \text{if } P_{send} = \langle p_s,\ I_s, O_s \oplus (o^{list}, [< \mathtt{o} >]), eqs_s \rangle \vdash_{(\mathbf{split}, (o^{list}, [<\mathtt{o}>]), \mathtt{Cons}\ o^i\ (o^{rest}, [<\mathtt{o}>]), \emptyset)} P'_{send} & & \end{array}$$

o^{list} = **Nil** means that the channel list identifier $(o^{list}, [< \mathtt{o} >])$ can be eliminated from the sender's interface and no new channels are introduced.

$$\begin{array}{ll} \langle p_s,\ In,\ \ Out \oplus \{(o^{list}, [< \mathtt{o} >])\}, & eqs \oplus \{o^{list} = [\,]\} \rangle \\ \vdash_{(\mathbf{split}, (o^{list}, [<\mathtt{o}>]), \mathtt{Nil}, \emptyset)} & \\ \langle p_s,\ In,\ \ Out, & eqs \rangle, \end{array}$$

$$\begin{array}{lll} & \mathcal{S} \oplus & \{P_{send},\ \ \langle (o^{list}, [< \mathtt{o} >]), \quad (p_s, p_r), \bot \rangle\} \\ \models & \mathcal{S} \oplus & \{P'_{send},\ \ \langle (o^{list}, [< \mathtt{o} >]), \quad (p_s, p_r), \mathbf{Nil} \rangle\} \\ \text{if} & P_{send} = \langle p_s,\ In, Out \oplus (o^{list}, [< \mathtt{o} >]), eqs \rangle & \vdash_{(\mathbf{split}, (o^{list}, [<\mathtt{o}>]), \mathbf{Nil}, \emptyset)} P'_{send} \end{array}$$

General channel structures

In the following, we will show how general data structures of channels can be transmitted. Like in the case of channel lists presented above, the action **split** specifies the data constructor that combines the substructures. In order to find out which cases represent 'recursive' splittings of structures and which 'termination cases', we now have to analyze the respective data structures.

Consider the (specialized) definition of a datatype `T`:

$$\texttt{data T}\ t_1 \ldots t_k = C_1\ t_{11} \ldots t_{1k_1} \mid C_2\ t_{21} \ldots t_{2k_2} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nk_n}$$

where $t_1 \ldots t_k$ represent fully annotated types[10]. The action

$$(\textbf{split}, s, C_i\ cs_{i1} \ldots cs_{ik_i}, \emptyset)$$

indicates that the channel structure s is split up into components $cs_{i1}, \ldots, cs_{ik_i}$ which can be transmitted via channel structures themselves. In general, for $j = 1, \ldots, k_i$ the following holds:

$$cs_{ij} \in \ \mathit{Channel\_Descriptor} \quad \text{(new channel descriptor)}$$

In the same way we equated `Cons` $(o^i, tg_o)(o^{rest}, type)$ and `Cons x xs` before, we now have to equate $C_i\ cs_{i1} \ldots cs_{ik_i}$ and $C_i\ x_{i1} \ldots x_{ik_i}$.

$$\begin{array}{l}
\langle p_{send},\ In,\ \ Out \oplus \{(o^{struct}, type)\},\ \ eqs \oplus \{o^{struct} = C_i\ x_{i1} \ldots x_{ik_i}\}\rangle \\
\vdash_{(\textbf{split},(o^{struct},type),\ C_i\ cs_{i1} \ldots cs_{ik_i},\emptyset)} \\
\langle p_{send},\ In,\ \ Out \oplus \{cs_{i1}, \ldots, cs_{ik_i}\},\ \ eqs \oplus \{c_{i1} = x_{i1}\} \oplus \ldots \oplus \{c_{ik_i} = x_{ik_i}\}\rangle, \\
\text{where} \\
cs_{ij} = (ci_{ij}, tag(t_{ij})), ci_{ij} \text{ new identifier}, ci_{ij} :: t_{ij}
\end{array}$$

$$\begin{array}{lll}
 & \mathcal{S} \oplus & \{P_{send},\ \ \langle (o^{struct}, type), (p_s, p_r), \bot\rangle\} \\
\models & \mathcal{S} \oplus & \{P'_{send},\ \ \langle (o^{struct}, type), (p_s, p_r), \texttt{C}_\texttt{i}\ cs_{i1} \ldots cs_{ik_i}\rangle, \\
 & & \ \ \langle (cs_{i1}, (p_s, p_r), \bot\rangle, \ldots, \langle (cs_{ik_i}, (p_s, p_r), \bot\rangle\} \\
\text{if} & P_{send} \vdash_{(\textbf{split},(o^{struct},type),\texttt{C}_\texttt{i}\ cs_{i1} \ldots cs_{ik_i},\emptyset)} & P'_{send}
\end{array}$$

Note the use of the constructor C_i with channel descriptors as its components, which is no type-correct Haskell expression. This "descriptor term" is however not used on the user-level, but forms only an intermediate representation of the structure. This representation will be transformed back into a type-correct Haskell expression upon receive, see Chapter 5.6.2. Nullary constructors do not lead to the creation of new channels. In this case the constructor, i.e. a term that does not contain any channel descriptors, is written to the contents of the channel structure, like in a regular **send** action.

The introduction of bypassing

If the identifiers of existing, but unused inputs occur in the definition of an output channel structure, these inputs can be redirected. When finding the defining equation $o^{struct} = C_i\ x_{i1} \ldots x_{ik_i}$ with names of existing channels or channel structures[11] as components of the right hand side, which are not needed elsewhere[12], these can safely be bypassed. This is signaled to the system by the set $\tilde{I}$. The number of channel (structures) to be newly introduced is decreased correspondingly.

[10] The bindings of type variables have to be used with the same annotations in all occurences.

[11] The $x_{il_1}, \ldots, x_{il_r}$ are meant to be identical to the corresponding $ci_{il_1}, \ldots, ci_{il_r}$. They are only synonyms introduced in order to simplify the notation in the following rules, not identifiers in the program.

[12] Here, we apply the same simple bypassing criterion as in the process creation (cf. Chapter 5.3.2).

$$\langle p_{send},\ In \oplus \tilde{I},\ \ Out \oplus \{(o^{struct}, type)\},\qquad eqs \oplus \{o^{struct} = C_i\ x_{i1} \ldots x_{ik_i}\}\rangle$$
$$\vdash_{(\mathbf{split},(o^{struct},type),\ C_i\ cs_{i1}\ldots cs_{ik_i},\tilde{I})}$$
$$\langle p_{send},\ In,\qquad Out \oplus \{cs_{is} \mid s \notin \{l_1,\ldots,l_r\}\},\ \ eqs \oplus \{\{ci_{is} = x_{is}\} \mid s \notin \{l_1,\ldots,l_r\}\}\rangle,$$

where
$\tilde{I} = \{cs_{il_1},\ldots,cs_{il_r}\} \subset Channel\_Descriptor$,
cs_{is} for $s \notin \{l_1,\ldots,l_r\}$ are new identifiers
$cs_{ij} = (ci_{ij}, tag(t_{ij})), ci_{ij} :: t_{ij}$
$x_{is} \equiv ci_{is}$ for $s \in \{l_1,\ldots,l_r\}$, and $ci_{il_1},\ldots,ci_{il_r}, \notin vars(eqs)$

Let $CS_{ij} = \langle (cs_{ij}, tg_{ij}), (p_j, p_s), cont_j\rangle$, for $cs_{ij} \in \tilde{I}$

$$\begin{array}{lll} & \mathcal{S} \oplus & \{P_{send}, \qquad \langle (o^{struct}, type), (p_s, p_r), \bot\rangle, CS_{il_1},\ldots,CS_{il_r}\} \\ \models & \mathcal{S} \oplus & \{P'_{send}, \qquad \langle (o^{struct}, type), (p_s, p_r), C_i\ cs_{i1}\ldots cs_{ik_i}\rangle, CS'_{i1},\ldots,CS'_{ik_i}\} \end{array}$$

where

$$CS'_{ij} = \begin{cases} \langle cs_{ij}, (p_j, p_r), cont_j\rangle, & \text{if } cs_{ij} \in \tilde{I}, \\ \langle cs_{ij}, (p_s, p_r), \bot\rangle, & \text{if } cs_{ij} \notin \tilde{I} \end{cases}$$

if $P_{send} = \langle p_s,\ I_s, O_s \oplus (o^{struct}, type), eqs_s\rangle \ \vdash_{(\mathbf{split},(o^{struct},type),C_i\ cs_{i1}\ldots cs_{ik_i},\tilde{I})}\ P'_{send}$

5.6.2 Receiving from channel structures

When receiving the contents of a channel structure, the receiver learns about changes related to the respective channel structure that have taken place in the system. The receiver receives a constructor term t_{desc} that contains the descriptors of new channels that have been created by **split**-actions (see Chapter 5.6.1). These channel descriptors are now introduced into the input interface of the receiver. By selecting only the ids of the descriptors in t_{desc} one obtains a data term t_{desc} of type *type* without the annotations. As is usual with channels, the original channel structure can be removed from the system after this receive operation. Note that the "descendants" of this channel structure do not notice this change until the receiver carries out **receive** actions on them.

$$\begin{array}{ll} & \langle p_{rec},\ In \oplus \{(c_{in}, type)\},\ \ Out,\ \ eqs\rangle \\ \vdash_{(\mathbf{receive},t_{desc},c_{in})} & \langle p_{rec},\ In \oplus NewIn,\qquad Out,\ \ eqs[c_{in} \mapsto t_{data}]\rangle \end{array}$$

$$\begin{array}{rcl} \text{where}\ \ NewIn & = & \text{set of channel descriptors contained in } t_{desc} \\ t_{data} & = & t_{desc}[(id_i, tag_i) \mapsto id_i \quad \text{for each } (id_i, tag_i) \in NewIn] \end{array}$$

$$\begin{array}{ll} & \mathcal{S} \oplus \ \{P_{rec}, \quad \langle (c_{in}, type), (p_{send}, p_{rec}),\ t_{desc}\rangle\} \\ \models & \mathcal{S} \oplus \ \{P'_{rec}\} \end{array}$$

if $P_{rec} \vdash_{(\mathbf{receive},t_{desc},c_{in})} P'_{rec}$

5.7 The Predefined Non-Functional Process `merge`

The instantiation rule for a `merge` process is similar to the instantiation of user defined processes. The list of inputs is a channel structure which has to be handled in the way shown in Chapter 5.6.1. The merging of input can start even if the structure of the input is not fully built up. The implementation will guarantee fairness, i.e. that neither the reading of values waiting in the individual inports is delayed indefinitely, nor the reading from the remaining channel list, if there is any. The input connected to the descriptor *inputList* is analyzed in the same way the input for a newly created user process is: If bypassing of this list is possible, the definitions will be $\tilde{I} = \{inputList\}$ and $I = \emptyset$, otherwise the other way round. Note that the bypassing of individual channels will be done later, namely upon split of *inputList*.

$$\begin{array}{l}
\langle p_{parent},\ In,\quad Out,\quad eqs \oplus \{o = \texttt{merge \# } inputList\}\rangle \\
\vdash_{(\textbf{create}, p_{merge}, I, O, \text{merge}, \tilde{I}, \tilde{O})} \\
\langle p_{parent},\ In \cup O \setminus \tilde{I},\ \ Out \cup I \setminus \tilde{O},\ \ eqs'\rangle \\
\text{where } I, O, \tilde{I}, \tilde{O} \text{ and } eqs' \text{ are defined as in the rules for process creation.}
\end{array}$$

The behaviour of the `merge` process is only defined at the system level. Its implementation is completely hidden and cannot be influenced by the programmer. This fact is reflected by the tag **merge** instead of equations. Let $P = \langle p, I \oplus \{c_{in}\}, \{out\}, \text{merge}\rangle$.

Every value waiting in any of the input channels is automatically transferred to the output channel:

$$\begin{array}{rl}
\mathcal{S} \oplus \{P, & \langle (c_{in}, \textbf{stream}), (p_{send}, p), l_{k+1} : rest\rangle, \\
 & \langle (out, \textbf{stream}), (p, p_{rec}), l_1 : \ldots : l_k : \bot\rangle\} \\
\models \mathcal{S} \oplus \{P, & \langle (c_{in}, \textbf{stream}), (p_{send}, p), rest\rangle, \\
 & \langle (out, \textbf{stream}), (p, p_{rec}), l_1 : \ldots : l_k : l_{k+1} : \bot\rangle\}
\end{array}$$

Input channels can be closed when their contents is used up. The merge process continues to merge the remaining inputs:

$$\begin{array}{rl}
\mathcal{S} \oplus \ \{ & \langle p, I \oplus \{(c_{in}, \textbf{stream})\}, \qquad \{(out, \textbf{stream})\}, \text{merge}\rangle, \\
 & \langle (c_{in}, \textbf{stream}), (p_{send}, p), [\,]\rangle\} \\
\models \mathcal{S} \oplus \ \{ & \langle p, I, \qquad \{(out, \textbf{stream})\}, \text{merge}\rangle\}
\end{array}$$

When there are no more input channels left, the output channel is closed and the `merge` process terminates:

$$\begin{array}{rl}
\mathcal{S} \oplus & \{\langle p, \emptyset, (out, \textbf{stream}), \text{merge}\rangle, \quad \langle (out, \textbf{stream}), (p, p_{rec}), l_1 : \ldots : l_k : \bot\rangle\} \\
\models \mathcal{S} \oplus & \{ \qquad \langle (out, \textbf{stream}), (p, p_{rec}), l_1 : \ldots : l_k : [\,]\rangle\}
\end{array}$$

5.8 Process Termination

If the output interface of a process becomes empty, the system[13] automatically detects its termination. Such a process $P = \langle p, \{i_1, \ldots, i_n\}, \emptyset, eqs\rangle$ is eliminated from the system together with all its input channels. Assume that I_p consists of the channel descriptors $i_1, \ldots, i_n$. As a consequence of the termination of P, the producers $p_1, \ldots, p_{n'}$ of these inputs[14], will remove the respective outputs from their interfaces:

$$\begin{array}{lll} \text{Let} & P = \langle p, \{i_1, \ldots, i_n\}, \emptyset, eqs\rangle \text{ and } C_p = \bigoplus_{k=1}^{n}\{\langle i_k, (p_k, p), cont_k\rangle\}, & \\ & \mathcal{S} \oplus \quad P \oplus \quad C_p \oplus \quad \bigoplus_{k=1}^{n'}\{\langle p_k, I_k, O_k, \qquad eqs_k\rangle\} & \\ \models & \mathcal{S} \oplus \qquad\qquad\qquad \bigoplus_{k=1}^{n'}\{\langle p_k, I_k, O_k \setminus I_p, \quad eqs_k\rangle\} & \end{array}$$

[13] Because all transitions described here are carried out by the system anyway, we do not need a special action for termination.

[14] The n inputs need not have distinct producers, i.e. $n' \leq n$.

Part II

Implementation

Chapter 6

Basic Requirements and Techniques

Level of abstraction. The design of a language, together with its semantics and implementation, always involves the introduction of different levels of abstraction. In the presentation of the language in Chapters 2 and 3, we have assumed the highest level of abstraction possible, which is the one that is most convenient to the programmer. In the operational semantics in Chapter 5, we have assumed a different level of abstraction, which is less abstract than the previous one, but still uses some very high level concepts.

In this part we will show that all the abstract concepts of Eden are designed in a way that leaves possibilities for efficient implementation. The point of view to be taken for the implementation is different from the above ones (see also Table 6.1). In fact, we will discuss the implementation of Eden on different levels of abstraction. We will start in this Chapter with a description of the general requirements to be fulfilled by any implementation, and afterwards discuss two different approaches to the implementation in detail, namely in Chapter 7 a prototype that works with static process networks, and in Chapter 8 the abstract machine DREAM.

Target architecture. The implementation described in this thesis has been developed for a system with distributed memory. In particular, we work with networks of workstations and an IBM SP/2. The latter is a multicomputer with an omega network. There, the latencies of messages are very high in comparison to computations and do not vary significantly in dependence of the communicating nodes. Experiments with benchmarks, among others, have been described by [HXA96].

Message passing using a standard library. In the parallel computing community, the library PVM[GBDJ94] and various implementations of the standard MPI[MPI94, MPI97] have gained wide acceptance. They can be called from C programs and offer portability. For this reason, our implementation also uses them. Details of MPI message passing routines will be presented in Chapter 7.2.3.

This selection of the parallel setting reflects the fact that Eden has been designed to be efficiently implementable on distributed memory machines. Nevertheless, it could be implemented on shared memory machines equally well. This fact is however not discussed further here.

6.1 Implementation requirements

In this section, we will discuss issues which *have to be solved* in any implementation in order to enable the system to handle all situations admitted by the language definition. This also includes measures which are virtually mandatory in order to make the system efficient.

6.1.1 Communication

Observable behaviour of different channels. For arbitrary data structures, we can not implement the observable behaviour of one-value channels by always sending one message, because the data structure could be either too large to be sent in one message, or too large to be buffered in the sender process. The same applies to streams with large elements. This means, that an implementation can be *forced* to transmit *more* messages than implied by the observable behaviour.

Optimal message size. In general, data to be transmitted should be separated into packets of a size that avoids the wasting of buffer space, that is efficient with regard to message passing and that allows interleaving of computations on the side of the sender and receiver as far as possible.

In particular, if the producer can not produce large amounts of data quickly enough, small messages will have to be used in order to prevent unnecessary delays or even deadlocks. This could be accomplished using a timeout, so that after a fixed time limit partially full packets are sent, or by a check for runnable threads: If no more output is available and no threads are running or runnable, send what is available at the moment.

Stream communication. Note that the use of a "stopping" mechanism like the ones mentioned above for stream message assembly will not be very satisfactory, because in cases where the one-by-one-transmission requirement is inherent to the system, the repeated use of such a mechanism would waste time in order to only re-discover a fact that was explicitly known to the programmer.

In cases where the use of a separate message for every list element would be inefficient, because the elements are generated quickly, the system can safely choose a larger message size, i.e. transmit *less* messages than implied by the observable behaviour. This can e.g. be done if on assembling a packet the system finds out that larger parts of the list are already available.

This means, the defaults and parameters to be used for stream communication and one-value communication are different: the observable behaviour defines the defaults and specific optimizations can be applied in many cases. Therefore it is vital that these two types of channels can be distinguished.

Latency hiding. An attempt to receive, i.e. read from, input that is not yet available results in blocking. It is important to hide communication latencies as far as possible by

1. interleaving different threads in a suitable way
2. defining separate receiving threads that handle the input information before it is needed by the computation threads (see also [Cha95][1]).

[1]The split-phase technique introduced in the STG_{MT} is not directly needed, because in Eden no

1:1 connections. The notion of a communication channel does not exist in MPI. However, it can be simulated by channel ids which are used as message tags.

The implementation has to guarantee that communication channels are 1:1 connections. For channels introduced on process creation, this can be achieved by simply assigning outports with unique tags to threads, which feed them. No computation thread knows the name of other outports. For dynamic channels, a runtime check is needed and the corresponding runtime error has to be "global"[2].

Buffer size. On the higher levels of abstraction, unlimited channel buffers (or receive buffers) are assumed. In the implementation, system messages have to be used in order to prevent buffer overflow. These messages affect message passing and thread scheduling, which is discussed below (cf. Chapter 8.6.4).

6.1.2 Topologies

Immediate creation of topologies. If multiple top-level process creations are carried out simultaneously, topologies can be built up directly, i.e. without resorting to the *run-time bypassing* mechanism modelled in the operational semantics. This is done, so to say, by transferring the actions taken by this mechanism from run-time to compile-time. This means, suitable compile-time analysis steps have to be established.

If all processes were created one after the other and connections would only be established between parents and children, process systems would have the structure of a tree. Connections in this tree could be optimized if processes did not access certain inputs or outputs, but simply pass them on to other processes. For every process abstraction, it can be checked which inputs are really consumed and which outputs are really generated inside the body. Those which aren't can be connected to other processes. They should be marked as "unused" at compile time. A suitable analysis step will inspect the process body in order to classify the names of in- and outports either as "produced" by this process, "consumed" by it, or otherwise as "unused", i.e. bypass-able.

The inputs and outputs to be analyzed in this way are the ones statically declared in the interface, plus the ones introduced by top-level process instantiations contained in the body.

Placement. In Eden, topologies are described on an abstract level. The actual placement of processes on processors is left to the system, because it depends on the load.

6.1.3 Threads and demand

Termination and number of outputs. Every process without any active outports will terminate. Consequently, it is important that the case of "one trivial output" `()` is interpreted as one and not zero outputs, in order to avoid premature termination of this process.

---

implicitly remote accesses to data are possible, but accesses to inports can be prepared by trying to receive the data as early as possible.

[2]Cf. the examples in Chapter 3.3.3 to see why it would be dangerous to raise a runtime error only in the process that tries to connect.

Throttling of threads. In the context of laziness versus strictness we have discussed, that a process transmits output spontaneously unless it is told otherwise, i.e. it receives a message requesting the retardation or termination of the communication. Such an interaction can be implemented easily, because Eden communication channels are directed and have a unique and known sender.

Creation of Processes. The MPI-1 standard[MPI94] does not cover the dynamic creation of processes. Consequently, most implementations of MPI do not offer functions that spawn processes dynamically. MPI-2[MPI97] however introduces it.

But MPI processes would be too "heavyweight" to be used as Eden processes. In UNIX systems, every MPI process is executed as a separate UNIX process, which involves considerable runtime expense. Efficient parallel programming with such processes entails the careful selection of processes for every particular combination of a program and a problem size. In order to remove this burden from the programmer, the Eden system performs scheduling of Eden processes itself. Only the scheduler processes will be modelled by MPI processes.

The different levels of abstraction used in the source language, the operational semantics, in the abstract machine DREAM (see Chapter 8) and in the DREAM-based parallel implementation (see Chapters 6.2.2 and 8) are summed up in Table 6.1 below.

level	language	communication (Chapter 6.1.1)	topology (Chapter 6.1.2)	threads + demand (Chapter 6.1.3)
Eden	Eden	streams and one-value channels	immediate creation	demand for outputs and top level process instant.
op. sem.	NKE	streams and one-value channels	one-by-one process creation	nondeterministic scheduling, process termination
DREAM	PEARL	primitive fct. sendVal, sendHead, closeStrm	compile-time bypassing	thread pool, process termination
impl.	C + MPI	message passing, limited buffers	placement	demand-driven scheduling and thread throttling

Table 6.1: Levels of abstraction in the source language, operational semantics, abstract machine (DREAM) and implementation: NKE stands for *normalized kernel eden* (see Chapter 4.3.2) and PEARL for *Parallel Eden Abstract Reduction Language* (see Chapter 8.2).

6.2 Implementation techniques

In this section, we will develop the design of Eden's implementation. In the previous section we have discussed implementation requirements in an abstract way. However, there are a number of questions which could not be answered without suitable practical experiments. In order to exploit insights gained by the first prototype(s) the best, the following hierarchical development is assumed. These implementations gradually include more features of Eden and more optimizations.

	system		**processes**	
Chapter	**topology**	**definition**	**threads**	**bodies**
6.2.1 and 7	static	predefined	single	purely functional
6.2.2 and 8	dynamic	in Eden	concurrent	all features
6.2.3	with runtime bypassing	in Eden	concurrent	all features

6.2.1 A static prototype

In the first prototype (see Chapter 7), topology descriptions directly implemented with MPI are combined with pure Haskell programs. This is useful to simulate what Eden programs do and to judge what performance could be achieved. It also helps to clarify the final structure of the parallel implementation.

As it is likely that a wide range of Eden processes will not *heavily* benefit from concurrency inside processes, this prototype can serve as a first approximation of the parallel implementation, although every process only contains one thread[3]. The only *coordination* aspect to be handled at this stage is communication. If the accessed data items are not yet received, the computation of the whole process is suspended, because there is only one thread of computation.

6.2.2 The implementation of DREAM

The abstract machine DREAM will be described in detail in Chapter 8. It introduces for every Eden process the following components: heap, global environment, inport table, a counter for blocked threads and a set of threads.

The inport table refers to a location in the heap, where received input is stored. This design doesn't give good support for run-time bypassing, because received messages are directly transferred to the heap. This causes difficulties, if the receiving process turns out to be not the consumer of the data. Run-time bypassing in any case will be so expensive that the overhead for the administration of bypassing may even outweigh the reduction in communication cost. Therefore this mechanism will not be considered in the implementation described here.

6.2.3 Implementation outlook

In the following, we will discuss ways to develop the above implementation further. We again adopt the classification into three areas underlying Chapter 6.1 and Table 6.1.

[3]But this (monadic) thread is quite powerful, because there can be several outputs, which are fed by in a sequence specified by the programmer.

Communication

Optimized treatment of channel structures. In [BKL97b] we have discussed cases where the size of a channel structure is known statically or at least on process creation. Such cases can be handled in an optimized way by a special transmission protocol. Such a protocol, e.g. for a channel list of length n, would choose an interleaved transmission of the sublists, which would work for channel lists where the length of the outermost list structure is finite and known on instantiation. This approach would also solve the efficiency problems resulting from superfluous concurrent threads.

Especially if skeletons convey information about the size of a channel structure, the expensive dynamic splitting process can be avoided.

Support for multicasting. In case a process uses an expression as output for multiple processes, this can either be seen as sharing or multicasting. Up to now we support only the interpretation as sharing of output among the corresponding threads. If large amounts of data are handled in this way, it may be much more efficient to perform one 1 : n communication (multicasting) operation instead of n point-to-point operations. This means, the implementation should recognize this situation and use collective communication. On the other hand, MPI's multicasting operations are synchronous. This may lead to problems if the receivers consume the data at different rates.

Topologies

Bypassing at runtime. In the implementation described above, we have restricted bypassing to the set of top-level process instantiations created simultaneously. In addition, bypassing can be possible for process instantiations embedded in other expressions, in the way the rule in Chapter 5.3.2 suggests it.

This more general form of bypassing can only be decided at the point of time when the second process is created. This means, one has to perform some automatic bypassing analysis at runtime, which efficiently exploits information about local access to data collected at compile time. The compile time analysis is an extension of the one described above, which analyzes *all* process instantiations.

For communication channels marked as "unused", connections between parent and child will no longer be established directly on process creation, but only "pointers", which later on will be returned by the real communication partner. The way the direct connection is established then resembles the handling of dynamic channels. Note however, that for channels to be bypassed on both sides, this reference has to be transferred from the real producer to the real consumer and back before the connection can be established. Only then the sender knows the real consumer of the data and the receiver knows the real producer.

Placement. The implementation could be developed further by offering more elaborate schemes for placement.

- Optimize for efficient communication, not only computation (distribution of the workload).

- Allow optional placement information specified by the programmer.

Threads and demand

Scheduling with priorities. In DREAM, we maintain a set of runnable threads which enjoy equal rights. In addition, we plan to provide special mechanisms for throttling threads that are "too eager". This means to influence the scheduler so that it assigns less time to such threads.

Should this kind of scheduling turn out to be too weak, a more global handling of scheduler priorities will be introduced. In a parallel setting, it really does matter which of two outports is serviced next, if one of them is consumed by a very slow process with an already full channel buffer and the other one by a fast one which is desperately starved for more input. In this case, a (wrong) local decision may degrade the overall performance drastically.

Future experiments will have to provide information about:

- the percentage of processes where such degrees of freedom exist. Naturally, it would not be worthwhile to introduce the additional overhead for the administration of changing priorities, if there hardly ever were a choice which thread to service next.
- the potential of global schemes in comparison to elaborate local schemes: Global schemes could adjust more quickly to the characteristics of the process system, while at the same time being less prone to over-reactions.
- the amenability of such characteristics of threads to automatic analyses or bookkeeping. Alternatively, optional annotations marking the "most important threads" could also be specified by the programmer.

Chapter 7

A MPI+Haskell Prototype

In this section, we describe a tool that gives Haskell programs the possibility to use the communication routines of MPI.

7.1 Motivation

The development of such a tool can be viewed from two different angles: Firstly, it can be seen as a first approach to the *implementation* of Eden. This view is suggested by the presentation in Chapter 6.2.1. Secondly, it can be seen as an independent case study or simulation that investigates parallel programming with Haskell plus skeletons in an *Eden-like manner*. This means, that although the implementation and the programming language are different, the kind of parallelism corresponds to that of Eden.

In the following, we will focus on the second viewpoint, because the reconciliation of MPI and Haskell is already a major step. It would not be possible to foresee all design requirements relevant for the parallel implementation of Eden. Therefore our aim is the construction of a small prototype with a structure that is only determined by requirements arising from the different characteristics of Haskell and MPI, and not from the implementation of (full) Eden.

7.2 How can Haskell programs use MPI?

In this section we will explain the different characteristics of Haskell and MPI with respect to the order of evaluation and the representation of data. This discussion will clarify what exactly has to be done in order to use the routines of MPI from a Haskell program.

7.2.1 Calling foreign language routines

MPI-routines can be called either from C or Fortran programs. So in order to use them from Haskell we can make use of **GHC**'s (Glasgow Haskell compiler[1]) C interface **Green Card**[NP96]. In the following we will sum up how C routines can be called from a Haskell program.

[1] see `http://www.dcs.gla.ac.uk/fp/software/ghc`

Defining a routine callable from Haskell. Green Card is a preprocessor that reads directives starting with % contained in Haskell programs. These directives are macros which are expanded prior to the compilation of the Haskell code by GHC. For the definition of a C routine, the following syntax is used: The type of the C routine is given after a **%fun** directive. This is followed by a **%call** directive, which specifies how the Haskell value `x` is converted into a C value. Analogously, a **%result** directive at the end of the C call defines the transformation of the C result back into the Haskell representation. Between these specifications, a block of C code defines the mapping of arguments to the result value, as shown in the example below:

```
%fun sin :: Float -> Float
%call (float x)
%{
     sinus = sin(x);
%}
%result (float sinus)
```

Syntax of a C call. In order to perform a call to a routine like the one above, the construct `_ccall_` is used. An example for a call to the above routine would be:

```
((_ccall_ sin (3.14 :: Float)) :: PrimIO Float)
```

The selection of the type of this operation will be explained in Chapter 7.2.2 below.

Marshalling. For many data types, the representations used in C and Haskell are not identical. For example, a string in C is internally represented by a null-terminated sequence of bytes and in Haskell by a `[Char]` in the heap. Not all data types exist in both languages. GHC only permits the transmission of certain data types which are instances of the type classes `CCallable` and `CReturnable`.

In the example above, we have used a predefined data type and specified the data interface scheme (DIS) `float` for it. For a user defined type `t` it would be necessary to specify two routines `marshal_t` and `unmarshal_t`, which define the conversion into this type and back.

7.2.2 Order of evaluation

When using a C routine, the programmer has to reason about side effects. If the routine called makes use of side effects, the C call has to be embedded into an external **state thread**[LJ94]. This would have been not necessary for the above example. But as indeed most C routines make use of side effects, all calls are embedded into the monad `PrimIO`. The state thread defines a sequence of actions. The monadic binding functions of `IO` and `PrimIO` are strict in their input state. Because there is demand for the result state of the thread, the whole thread will be evaluated[2]. When calling a C routine, unboxing (see [PL91]) is performed on the arguments of the routine.

[2]See [Pri97] for a discussion why state threads and thereby monadic programming in the whole user program have been chosen.

7.2.3 Communication concepts in MPI

In this section, we will explain the communication routines of MPI and how they *could be used* from a Haskell program. In the next section, we will present a more general environment for doing this.

The components of MPI. The message passing interface standard defined in 1994 [MPI94] is now referred to as "MPI-1" because in the meantime, the successor standard [MPI97] has been defined (see also [GGHL+96]). In the present work, we use MPI-1, because no implementation of MPI-2 on our target machines is available.

The central routines of MPI-1 are shown below (cf. [GLS94, Table 2.1]). The C routines all receive a number of arguments (which are not shown due to space limitations) and return an integer error code.

`MPI_Init`	initialize MPI
`MPI_Comm_size`	find out how many processes there are
`MPI_Comm_rank`	find out which process I am
`MPI_Send`	send a message
`MPI_Recv`	receive a message
`MPI_Finalize`	terminate MPI

Sequence of actions. When using lazy evaluation, the sequence of actions taken is driven by the demand for output. This is appropriate for purely functional computations, but not for computations which indirectly depend on the side-effects of other actions. Such computations could of course be transformed into functional actions with an "artificial" dependency on the "result" of the side-effecting operations, but in the context of the MPI routines this solution would be very inconvenient.

Note that the dependencies between different MPI calls are very numerous: Firstly, for an arbitrary call to be successful, `MPI_Init` has to be completed before. Secondly, for many communication calls a certain sequence has to be guaranteed in order to prevent deadlocks, e.g. for a pair of processes exchanging messages, it is vital that both of them first carry out the send operation and afterwards the receive operation.

Structure of a program with C calls. The above requirements will be enforced by using an appropriate state thread, into which functional computations will be embedded in the following way:

```
...
returnPrimIO (f e1 .. em) >>= \result ->
...
```

A functional program written in the monadic style can be separated into an input- , a computation- and an output phase. In order to enforce the requirements arising from MPI communication, we have to extend this structure:

```
      Haskell:      Input ;                Computation;                    Output
        ⇓
  Haskell + MPI:    Input ;    Init ;      Computation;      Exit ;        Output
                               MPI_Init                   MPI_Finalize
```

Point-to-point communication. The routines `MPI_Send` and `MPI_Recv` realize 1:1 communication. There are synchronous and asynchronous variants of these functions. In this context, only the synchronous versions can be used, because for the passing of values between Haskell and C, the read or write operations are assumed to be completed. The implementation of asynchronous communication with these synchronous routines will be explained in Chapter 7.3.1 below.

Collective communication. MPI provides a number of routines for 1:n and n:1 communication. There are routines for the movement of data (broadcast, scatter, gather), which can be used here for the distribution of input data and for the collection of output data. In addition, there are routines for performing a collective reduction operation and for barrier synchronization.

All these routines work synchronously, i.e. they can only complete successfully if all processes in the respective group of processes call it. They can be used from Haskell, provided that the programmer ensures that all processes involved really *carry out* the respective call.

7.3 The design of the MPI + Haskell tool

Overview. The discussion above has made clear the basic phases that Haskell programs have to execute in order to communicate via MPI. This basic scheme can also serve as a guideline for the design of our tool.

7.3.1 The type class `Transmissible`

In the previous section, we have mentioned a number of steps that have to be taken when using MPI communication from a Haskell program. In the following, we will explain how this handling can be supported by using a Haskell type class.

We define a type class `Transmissible` with overloaded operations for sending and receiving. The specific methods will transmit values of the corresponding type. Arbitrary data types can become instances of this class, if specific functions for their transmission are defined. The definition of these functions corresponds to the definition of a communication protocol. The definition of class `Transmissible` is shown in Figure 7.1 below.

```
class Transmissible a where
                      --  context  val   dst/src  tag     length  result

   send_to_with       ::           a ->  Int ->   Int ->          PrimIO Int
   send_c_to_with     ::  Int ->   a ->  Int ->   Int ->  Int ->  PrimIO Int

   recv_from_with     ::                 Int ->   Int ->          PrimIO a
   recv_c_from_with   ::  Int ->         Int ->   Int ->          PrimIO a
```

Figure 7.1: Class `Transmissible` (shortened)

We work with a so-called *context*, which indicates whether a value occurs isolated or inside a sequence of values. The above class, together with a number of instances and

auxiliary definitions, is provided in a file `trans.hs`, that has to be included by Haskell programs that communicate via MPI.

Instances for primitive types. The transmission of an integer value can either occur separately (context 0) or at the beginning, in the middle or at the end of a sequence of values (context 1-3), see Figure 7.2:

```
instance Transmissible Int where
   send_to_with val dst tag = send_c_to_with 0 val dst tag 0
   send_c_to_with context val dst tag length = case context of
    0 ->  _ccall_ send_s_int  val dst tag         -- send val directly
    1 ->  _ccall_ sl1_int     val dst tag length  -- alloc.  buff., store val
    2 ->  _ccall_ sl2_int     val dst tag         -- store val
    3 ->  _ccall_ sl3_int     val dst tag         -- send, deallocate buffer
   recv_from_with src tag = recv_c_from_with 0 src tag
   recv_c_from_with context src tag = case context of
    0 ->  _ccall_ recv_s_int  src tag             -- receive directly
    1 ->  _ccall_ rl1_int     src tag             -- allocate, receive all
    2 ->  _ccall_ rl2_int     src tag             -- return val
    3 ->  _ccall_ rl3_int     src tag             -- deallocate buffer
```

Figure 7.2: Instance `Transmissible Int`

Instances for other primitive types can be defined in a similar way.

Instances for lists. A list type `[a]` can become an instance of class `Transmissible`, provided that the element type `a` also belongs to this class. In this way, an *inductive definition* of a communication protocol is given. Parts of the `Transmissible` instance `[a]` are shown in Figure 7.3 below. Function `send_c_to_with` scrutinizes the value to be sent. Empty lists are assigned code `1` and nonempty ones (`v:vs`) code `2`. In the latter case, the length of the list is communicated and afterwards the head of the list is sent with context `1`, by evaluating `send_c_to_with 1 v dst tag length`. Note that this refers to the `send_c_to_with` method for the *element* type, which is also the reason why `Transmissible a` is a prerequisite for `Transmissible [a]`. Moreover, an auxiliary function `send_c_tw` handles the tail `vs` in an analogous way. It also belongs to `Transmissible` and is not shown here, see [Pri97] for details.
The above packaging of data by means of a list context can be adopted for general user defined data types. As a guideline for the definition of instances of `Transmissible`, this method should be employed for composite types with several components.

Due to the fact that the complete structure is evaluated prior to sending, this method can only be used for *finite* lists. In the following subsection we will discuss the transmission of *streams*.

Stream communication. If one wants to receive infinite lists, one can use the following trick in order to prevent an update and the subsequent sharing of the input stream (see Figure 7.4): `input_str` is defined by the auxiliary function `i_str`, which continually performs a receive operation and afterwards calls itself again. The argument `n` is never

```
instance Transmissible a => Transmissible [a] where
   send_to_with val dst tag = send_c_to_with 0 val dst tag 0
   send_c_to_with context val dst tag _ = case context of
     3 -> returnPrimIO 0
     _ -> case val of
        [] -> <send 1>
        (v:vs) -> <send 2>
                  <send length>
                  send_c_to_with 1 v dst tag length >>
                  send_c_tw vs dst tag >>= \result ->
                  returnPrimIO result
```

Figure 7.3: Instance `Transmissible [a]` (shortened)

used, its only purpose is the prevention of sharing. Without it, after the first evaluation of `i_str`, an update would be performed and there would no longer be demand for the side-effect, namely the receive operation. In this case, an infinite list would be returned, where all elements were equal to the first value received. The receive operation used will be explained in Chapter 7.3.2 below.

```
input_str ::  [Int]
input_str = i_str 0

i_str ::  Int -> [Int]
i_str = \n -> (unsafePerformPrimIO (((recv_from (my_id-1))::PrimIO Int))
               :(i_str n))
```

Figure 7.4: Expression that reads a stream (infinite list) of `Int` values

7.3.2 A library of predefined routines

The tool provides two files with C routines that can be used from Haskell programs. The file `wrap_mpi.c` contains the routines `init_phase` and `exit_phase` which initialize resp. finalize MPI and in addition handle communication buffers.

In `prim_mpi.c`, send and receive routines for the primitive data types are defined. For a type `<type>` out of `Char`, `Int`, `Double`, etc. the following routines are provided:

The routines `send_s_<type>` and `recv_s_<type>` send or receive a *single* value of the specified type. In addition, we provide the sets of routines `sl{1,2,3}_<type>` and `rl{1,2,3}_<type>` which support the buffered transmission of *sequences* of values. The numbers indicate the *context* of the operation, cf. Chapter 7.3.1.

The sending of a list of values is started by routine `sl1_<type>`. A call to this routine leads to the allocation of a buffer for the list elements. The first element, which is passed as the argument of the routine, is stored in this buffer. Subsequent calls to `sl2_<type>` transfer further list elements into this buffer. Finally, `sl3_<type>` writes a last element to the buffer, transmits the whole contents of the buffer and deallocates it.

The reception of list values is organized analogously. The operation `rl1_<type>` allocates memory for the list of values to be received and receives the whole list and returns the first element as its result. Correspondingly, the following elements can be received using `rl2_<type>`. The final value is received by a call to `rl3_<type>`, which at the same time discards the buffer.

7.3.3 The programming notation

MPMD style. In the following we will present a framework that allows the flexible and intuitive definition of communicating processes written in Haskell. The programs defining the individual processes will be written in an MPMD style and combined into a single file that lists the code to be executed by the different "nodes":

```
% NumProcs N ..
% SharedDecls
   % include ''trans.hs''
   ...
% Proc 0
   main = ...
% Proc 1..(N-2)
   main = ...
% Proc N-1
   main = ...
```

The preprocessor `splitup`. Our tool uses a Perl script called `splitup` in order to generate `N` Haskell programs with the suitable definitions from a specification like the one above. In addition, it inserts functions for finding out the size of the process group and its own process id.

The notation. The above scheme already shows a number of components of the notation used by splitup. The full grammar of this notation is as follows:

Using the components presented in this section, parallel programs can be developed in a skeleton-like way. The next section will give examples of such skeletons.

7.4 Examples of skeletons

7.4.1 Pipeline

In Figure 7.4, we have shown code usable for a process that reads a stream of `Int`. We can now use the above notation in order to define a pipeline with `N` processes by inserting this code into the following skeleton. The code for the internal pipeline stages, i.e. processes `1..(N-2)` are shown in full. The initial and final stages can be defined in an analogous way and are therefore omitted here.

7.4.2 Tree

The following example shows the definition of a tree of processes.

program	*prog*	→	*numprocs section*$_1$ ... *section*$_n$	$n \geq 1$
section	*section*	→	*shared*	
		|	*proc*	
N processes	*numprocs*	→	`% NumProcs` *var integer nl*	
shared code	*shared*	→	`% SharedDecls` *nl hs_inc*$_1$ ... *hs_inc*$_n$	$n \geq 1$
code for process	*proc*	→	`% Proc` *ranges nl hs_inc*$_1$ ... *hs_inc*$_n$	$n \geq 1$
	ranges	→	*range*$_1$, ... , *range*$_n$	$n \geq 1$
	range	→	*arith_expr*	
		|	*arith_expr* .. *arith_expr*	
Haskell + include	*hs_inc*	→	Haskell code	
		|	`%include ''` *dateiname* `''` *nl*	
newline	*nl*	→	$\backslash n$	
expression over N	*arith_expr*	→	...	
variable names	*var*	→	`a` | `b` | ... | `z` | `A` | `B` | ... | `Z`	
(comment	*comment*	→	`#` *string nl*	)

Figure 7.5: Grammar of `splitup` notation "`dcl`"

```
% NumProcs N ..
# ----------------------------------------------------------------------
% SharedDecls
% include ''trans.hs''
computationFct ::  ....  -> [Int]
computationFct ...  = ...
# ----------------------------------------------------------------------
% Proc 0
...
# ----------------------------------------------------------------------
% Proc 1..(N-2)
input_str ::  [Int]
input_str = ...  -- see above
traversalFct ::  [Int] -> PrimIO ()
traversalFct [] = returnPrimIO ()
traversalFct (v:vs) = send_to_with v (my_id+1) 0 >>
              traversalFct vs

main = primIOToIO $
      ((_ccall_ init_phase) ::  PrimIO Int) >>
      traversalFct (computationFct input_str ...)  ...  >>
      ((_ccall_ exit_phase) ::  PrimIO Int) >>
      returnPrimIO ()

# ----------------------------------------------------------------------
% Proc (N-1)
...
```

Figure 7.6: Skeleton for a pipeline that communicates streams of `Int`

Example 7.1

```
% NumProcs N 7
# ---------------------------------------------------------------------------
% SharedDecls
% include "trans.hs"
topNbr   k = if k /= 0 then (k - 1) `div` 2 else error "top 0"
leftNbr  k = let nbr = 2 * k + 1
             in if nbr <= n then nbr        else error "left LEAF"
rightNbr k = let nbr = 2 * k + 2
             in if nbr <= n then nbr       else error "right LEAF"

# -- root node: 2 neighbours (left, right) ----------------------------------
% Proc 0
main = ((_ccall_ init_phase) :: IO Int)           >>
       perform_bisort_root len 0                  >>
       ((_ccall_ exit_phase) :: IO Int)           >>
       return ()
# -- inner nodes: 3 neighbours (only present for N >= 7) --------------------
% Proc 1..(((N-1)/2) - 1)
main =  ((_ccall_ init_phase) :: IO Int)           >>
        (perform_bisort_node (len `div` 2) 0 )     >>
        ((_ccall_ exit_phase) :: IO Int)           >>
        return ()

# -- leaves: 1 neighbour (top)-----------------------------------------------
% Proc ((N-1)/2)..(N-1)
main = ((_ccall_ init_phase) :: IO Int)            >>
        traversalFct (leaf_agent input_str)        >>
        ((_ccall_ exit_phase) :: IO Int)           >>
        return ()
```

The above program is a shortened version of the one which has been used for runtime measurements with a bitonic merge sort algorithm in [BL98]. The definitions of `input_str` and `traversalFct` are omitted here, because they are identical to the ones in the pipeline. The functions `perform_bisort_root` and `perform_bisort_node` perform a sequence of computation and communication steps. The development of the `leaf_agent` is analogous to the pipeline stage shown before. ◁

In [Pri97], more skeletons can be found. Among them are skeletons for general grids and graphs and skeletons for reactive systems.

7.5 Discussion

7.5.1 Performance analysis

The Eden-prototype version of the bitonic merge sort program shown in [BL98] has been implemented using the tree skeleton shown in Example 7.1.

The following table contains runtimes for a tree with 7 SUN SPARCstations 10 con-

nected by Ethernet using the Glasgow Haskell compiler 2.09 and LAM-MPI[3] under Solaris. For the times shown below, LAM was run with the option `c2c`, which uses direct communication between the MPI processes without daemons in between. The use of this option decreased the runtimes by about 15% on the average. The runtimes shown are the CPU times delivered by the Haskell library function `getCPUTime`.

length of list	512	1024	2048
sequential runtimes (sec)	0.34	0.83	1.94
7 procs runtimes (sec)	0.12	0.22	0.42

Taking into account that these figures have been produced using a slow communication network and that a tree structure cannot be mapped to an Ethernet without contention, the speedup over the sequential version is a quite impressive result.

In contrast to this, a corresponding version with implicit parallelism did not even display speedups on a shared memory workstation, because it built up and abandoned the process tree several times (cf. [BL98]).

This paper contains a comparative case study of different versions of a parallel bitonic merge sort algorithm. The `dcl` version using a stable process network achieved much better results than a version with implicit parallelism that works with short-lived processes. In this way it is shown that explicit and stable networks of processes are desirable from an implementation point of view.

Portability. For this tool, only features of the MPI-1 standard have been used. Consequently, it should be easily possible to use it on all platforms for which an implementation of MPI (and a port of the GHC, of course) exist.

Topologies. Our prototype allows the description of skeletons in the sense that we can select a number of processes and specify which of them communicates with which of the other processes. We can't select directly the actual layout of this graph on the parallel machine or network, but the MPI environment allows the explicit selection of machines for the different executables. The skeletons could be extended in order to create the respective MPI files as well and accept a specification of *topology* information, provided that runtime experiments indicate that the physical layout does matter in a message-passing environment with MPI.

Presumably there is not much variation in the message passing times between different *pairs* of nodes. The High Performance Switch of the SP/2 allows the simultaneous routing of multiple messages. The *overall* communication topology may be more important, i.e. the number of messages the system can handle at the same time, in dependency of the sets of senders and receivers. This means, care should be taken that "communication channels" frequently used at the same time should be conflict-free, i.e. not delay each other.

Future experiments have to clarify whether this aspect can affect performance noticeably on the target machines used or whether only a small number of pathological communication patterns will degrade performance.

Granularity. It is obvious that programming with the `dcl` tool is much more complicated and error-prone than programming in Eden. Presumably a carelessly written `dcl` program

[3]see `http://www.osc.edu/lam.html`

will perform worse than any "equivalent" Eden program. This is due to the lower level of abstraction, that gives the programmer the possibility to make more serious mistakes.

On the other hand, nearly optimal programs can be expressed with `dcl`. In this way, our tool marks the limit of the performance that can be achieved using explicitly parallel functional programming. This particularly applies to granularity. Optimal performance can be achieved if the programmer can estimate the process *granularity* and selects the placement appropriately.

Alternatively, optional placement directives could be introduced. Other ways for the management of granularity information are discussed in [Loi97a].

7.5.2 Implicit versus explicit parallelism

Our prototype can be viewed as a "representative" of the explicitly parallel programming model underlying Eden, or an even more explicit one (see above). This permits a comparison to the "semi-explicit" approach taken in Glasgow parallel Haskell. For instance, the paper[HLT+97] describes an algorithm that calculates bowings, where the following obstacles to (implicit) parallelism are encountered:

- the granularity of the threads has to be controlled and the most computation-intensive ones have to be created first. This is necessary because the threads vary greatly in duration, and the ones that form the critical path of the overall computation have to be started first in order to avoid long waiting times near the end of the computation. Note that GUM neither supports thread migration nor preemption.

- The placement of the threads has to be controlled in order to prevent more than one computation-intensive thread from being assigned to the same processor. In this particular application, it was useful to restrict the thread pool size to one. This only works in cases where all the threads are created in the beginning.

Especially the second measure described above is a very application-specific heuristic. This illustrates that for general programs it may be helpful to make more aspects of a programming language explicit. In this way, one could start programming with the least explicit version and gradually descend to more explicit versions, if required due to performance issues. For experimenting with these, our tool provides a useful testbed. Suitable simulations using our `dcl` tool form the first step in order to decide whether it would be useful to make these aspects more directly visible in the source language.

7.5.3 Relationship to Eden

The convincing results make our prototype interesting in its own right. Moreover, it shows that the explicit handling of processes in Eden is desirable from an implementation point of view. Now that it is implemented, a number of aspects have to be investigated in order to relate it to Eden and to its full parallel implementation:

Preparing the ground for the parallel Eden implementation. Parts developed here can either be reused or at least serve as groundwork for the parallel implementation of Eden. The C routines for communication and the class `Transmissible` are useful starting points for the development and test of the primitive communication functions used in DREAM (see Chapter 8).

Case study: structure of efficient parallel programs. This tool can help to gain information about the source language. It can help to clarify how efficient programs should look like. Additionally it can be used to decide which additional features could improve the performance of Eden programs, by simulating their existence in a `dcl` program.

Performance comparison. Although based on a different source language, this tool enables us to write programs that *mimic the runtime behaviour* of Eden programs. In exposing much more of the implementation details of a parallel computation than Eden programs do, it can be used to simulate the behaviour of Eden programs in combination with a variety of optimizations in the implementation.

When performing such a comparison, one should keep in mind the different implementation techniques underlying this tool and the Eden implementation. In particular, measurements have to be carried out that compare the two systems in order to clarify and quantify the performance differences between them. There are aspects which make this prototype (potentially) *more* efficient than the Eden implementation, such as static communication topologies and simpler run-time structures due to the absence of thread scheduling.

On the other hand, there are also aspects that make it (potentially) *less* efficient than the Eden implementation, above all the single-threading restriction and the resulting inability to hide communication latencies. Based on such results, the performance of possible optimizations in the Eden implementation could be *predicted* by simulating them with the tool explained here.

The role of processes. One of the insights gained from this case study is the usefulness of explicit processes and stable process networks.

For this prototype, we have identified the processes defined on the level of the source language with the processes of MPI. As the MPI implementation used did not support dynamic process creation, we restricted ourselves to static systems of processes. Note that every particular computation can be simulated with a system that supports only static process management.

For the parallel implementation of Eden, we will abandon this identification of user-level processes and MPI processes, even if the parallel platform had a suitable implementation of MPI-2. The reason for this is the "heavy weight" of MPI processes. The definition of parallel processes that could be usefully executed on the particular underlying machine would be a too hardware- and load-specific task.

The view of processes taken for this tool somehow defines a limit for a parallel program: The programmer can define processes and their placement freely. If this mapping turns out to be optimal, the runtime efficiency is high. For Eden's parallel implementation, we will however assume several Eden processes to be executed by one MPI process in order to make programming easier.

Chapter 8

The Abstract Machine DREAM and its Implementation

8.1 Introduction

In this chapter we will show how the distributed implementation of Eden (introduced in Chapter 6.2.2) can be developed in the same modular way as the language definition. It incorporates a parallel runtime system that is specifically tailored for Eden[1]. Following the lead of other functional language implementations, full Eden is first desugared and translated into a core language called PEARL[2]. (Parallel Eden Abstract Reduction Language), then this language is conceptually interpreted by a distributed abstract machine called DREAM (DistRibuted Eden Abstract Machine), see Chapter 8.3.

PEARL is finally compiled to C code with MPI calls. An important design decision for the present implementation has been to use the GHC as a basis, and to reuse of it as much as possible to save development effort and to achieve high efficiency. In order to meet this goal, we have defined PEARL as an extension of the STGL (Spineless Tagless G-machine Language) [Pey92] underlying the GHC, and DREAM as an extension of the STGM (Spineless Tagless G-Machine).

8.2 PEARL

PEARL is an extension of the core language STGL of the Glasgow Haskell compiler for the kernel constructs of Eden. Figure 8.1 shows the syntax of STGL as it has been defined in [Pey92]. An STGL program is a non-empty sequence of bindings of variables to lambda forms. A lambda form $vars_f \backslash \pi\ vars_a$ `->` $expr \in Lfs$ is a lambda abstraction $\backslash vars_a$ `->` $expr$ which additionally contains information about the free variables $vars_f \subseteq Var$ and a flag π which indicates whether the lambda form, or to be precise, a closure with this lambda form, must be updated after its evaluation or not. The body expressions of lambda forms may be literals, applications, local definitions or case expressions. In the following, we denote the set of STGL-expressions by $Expr$, the set of alternatives of `case` expressions by $Alts$ and the set of typed data constructors by Γ.

[1] This also puts it in contrast to a previous uniprocessor prototype[BKL97a].

[2] which is not to be confused with the scripting language *Perl*

prog	$\rightarrow$	*binds*	
binds	$\rightarrow$	$bind_1$; ... ; $bind_n$	$n \geq 1$
bind	$\rightarrow$	*var* = *lf*	
lf	$\rightarrow$	$vars_f \backslash \pi$ $vars_a$ `->` *expr*	
π	$\rightarrow$	$u \mid n$	
expr	$\rightarrow$	*literal*	
	$\mid$	*constr atoms*	constructor application
	$\mid$	*primOp atoms*	primitive application
	$\mid$	*var atoms*	application
	$\mid$	`let` *binds* `in` *expr*	local definition
	$\mid$	`letrec` *binds* `in` *expr*	local recursive definition
	$\mid$	`case` *expr* `in` *alts*	case expression
vars	$\rightarrow$	$\{var_1, \ldots, var_n\}$	$n \geq 0$, variable list
atoms	$\rightarrow$	$\{atom_1, \ldots, atom_n\}$	$n \geq 0$, atom list
atom	$\rightarrow$	*var* $\mid$ *literal*	
literal	$\rightarrow$	`0#` $\mid$ `1#` $\mid$ ...	primitive integers
primOp	$\rightarrow$	`+#` $\mid$ `-#` $\mid$ `*#` $\mid$ ...	primitive operations
alts	$\rightarrow$	$calt_1$; ... ; $calt_r$; *default*	$r \geq 0$, algebraic alternatives
	$\mid$	$palt_1$; ... ; $palt_n$; *default*	$n \geq 0$, primitive alternatives
calt	$\rightarrow$	*constr vars* `->` *expr*	
palt	$\rightarrow$	*literal* `->` *expr*	
default	$\rightarrow$	*var* `->` *expr*	
	$\mid$	`default` `->` *expr*	

bind	$\rightarrow$	$vars_o = var_p$ `#` $\{var_1 \mid\mid \ldots \mid\mid var_n\}$	**process instantiation**
lf	$\rightarrow$	$vars_f$ **process** $vars_a$ `->` *pBody*	**process abstraction**
pBody	$\rightarrow$	`letrec` *binds* `in` *parExpr*	**process body**
	$\mid$	*parExpr*	
parExpr	$\rightarrow$	$\{expr_1 \mid\mid \ldots \mid\mid expr_k\}$	$k \geq 1$
expr	$\rightarrow$	`new` (var_n, var_c) *expr*	**dynamic channel handling**
	$\mid$	var_n `!*` $expr_1$ `par` $expr_2$	
primOp	$\rightarrow$	**sendVal** $\mid$ **sendHead** $\mid$ **closeStrm**	**primitives for communication**
	$\mid$	**split**	

Figure 8.1: Syntax of STGL (above) and PEARL extensions (below)

The PEARL extensions (see Figure 8.1) concern bindings, lambda forms, expressions and primitive operators. In addition to the normal binding of STGL we introduce new bindings which define process instantiations. Process instantiations will only occur in the binding part of `let` and `letrec` expressions.

Lambda forms in PEARL can be either process abstractions or STGL lambda forms defining functions. Process abstractions are lambda forms with the tag `process` and a process body that evaluates to an expression $\{expr_1 || \ldots || expr_k\}$ defining parallel threads for the process outports. The two lists of variables $vars_f$ and $vars_a$ contain the free variables of the process body and the inports of the process abstraction, respectively.

As process instantiations and abstractions are only handled as bindings and lambda forms, and not as special expressions, new expressions are merely introduced for dynamic channel handling. PEARL contains `new` *and* `par` *expressions* for the creation and use of dynamic channels.

Communication using simple channels. We use three new *primitive operations* in order to manage the sending of evaluated values via channels. The function **sendVal** is used for sending a single value, **sendHead** sends the next value of a stream and **closeStrm** closes a stream. These form the basis of the predefined function `sendChan` with type `a -> ()` shown below, which transmits values on a channel.

```
sendChan  l = case l of {Nil         -> closeStrm;
                         Cons {y,ys} -> force y `seq` sendHead y `seq`
                                        sendChan ys;
                         default     -> force l `seq` sendVal l }
```

`sendChan` uses an overloaded function `force :: a -> a` to reduce expressions to normal form. The function `seq`, as predefined in Haskell 1.3 onwards, evaluates its first argument and then returns its second argument.

```
force x = force' x `seq` x
force' (C a1 ... an) = force a1 `seq` ... `seq` force an `seq` ()
```

The monomorphic instances of `force` will be derived by the compiler for any data type that may be sent. The compiler will also insert an application of `sendChan` to the expressions to be evaluated by a thread.

PEARL has been chosen to be as close as possible to STGL in order to reuse the STGM for the sequential code of Eden. Note that the only new constructions to be taken care of by DREAM are process abstractions, instantiations, the constructs `new` and `par` as well as the new primitives **sendVal**, **sendHead** and **closeStrm**.

Compiling Eden into PEARL: We extend the transformation of Haskell into STGL by rules for transforming the Eden constructs into the corresponding PEARL expressions. Only process abstractions and instantiations are transformed in a special way. The sending of data is made explicit in PEARL by inserting calls of the function `sendChan` at the positions of output expressions in process abstractions and instantiations. Moreover multithreading is indicated by the construct $\{\ldots || \ldots || \ldots\}$.

The transformation of process abstractions into lambda forms ensures that they can only appear as right hand sides of bindings. A process abstraction of the following form

$$\texttt{process}\ (var_1, \ldots, var_n)\ \rightarrow (out_1, \ldots, out_m)\ \texttt{where}\ equations$$

is translated into

$$
\begin{array}{l}
\texttt{let}\ v_{new} = \ \{frees\}\ \backslash\texttt{process}\ \{var_1, \ldots, var_n\} \rightarrow \\
\qquad\qquad \texttt{letrec}\ \ equations' \\
\qquad\qquad\qquad\qquad v_{new1} = out'_1 \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad v_{newm} = out'_m \\
\qquad\qquad \texttt{in}\ \{\texttt{sendChan}\ v_{new1} || \ldots || \texttt{sendChan}\ v_{newm}\} \\
\texttt{in}\ v_{new}
\end{array}
$$

where the v_{newj} are new variables and out'_j and $equations'$ are the transformed output expressions and equations, respectively.

As process instantiations are special bindings in PEARL, they are abstracted out of expressions and put into local declarations (which float outwards in the optimization passes used in the GHC). An instantiation of the above process abstraction:

$$pabs\ \#\ (in_1, \ldots, in_n)$$

is transformed into

$$
\begin{array}{l}
\texttt{let}\ \ v_{new1} = in'_1 \\
\qquad\ \ \vdots \\
\qquad\ \ v_{newn} = in'_n \\
\texttt{in let}\ \ v_{inp1} = \texttt{sendChan}\ v_{new1} \\
\qquad\qquad\ \ \vdots \\
\qquad\qquad\ \ v_{inpn} = \texttt{sendChan}\ v_{newn} \\
\texttt{in let}\ \ (newOut_1, \ldots, newOut_m) = pabs\ \#\ \{v_{inp1} || \ldots || v_{inpn}\} \\
\texttt{in}\ (newOut_1, \ldots, newOut_m)
\end{array}
$$

where the v_{newi}, v_{inpj}, and $newOut_k$ are new variables and in'_l are the transformed input expressions of the child process.

In order to pass Eden programs through the GHC front end, all Eden constructs have been embedded into Haskell using dummy functions which impose the correct types on them. Haskell's type inference will thus check the type consistency of process abstractions and instantiations. Only for the extended bindings in process bodies additional rules have been inserted in the compiler front end passes. The transformation of the Eden constructs into PEARL is done at the end of the front end passes.

Example 8.1

The following Eden program specifies a parallel mergesort algorithm:

```
sortNet = process list -> sort list
          where sort []  =  []
                sort [x] =  [x]
                sort xs  =  merger # (sortNet # l1, sortNet # l2)
                            where (l1,l2) = unshuffle xs
                unshuffle []      = ([],[])
                unshuffle [x]     = ([x],[])
```

```
unshuffle (x:y:t) = (x:t1,y:t2)
                    where (t1,t2) = unshuffle t
merger = process (s1,s2) -> smerge s1 s2
         where smerge []    l     = l
               smerge l     []    = l
               smerge (x:l) (y:t) = if x<=y
                                    then x:smerge l (y:t)
                                    else y:smerge (x:l) t
```

The sorting net unfolds only in case the input list has at least two elements. If we had lifted the process instantiations to the top level by an equation of the following form, an infinite process system would have been produced:

```
out = merger # (sortNet # l1, sortNet # l2)
      where (l1,l2) = unshuffle list
```

This will be translated into the following PEARL program where the bindings for `force` and `sendChan` are provided in the global environment:

```
sortNet = {sendChan,sortNet} \process {list} ->
    {let merger = {sendChan} \process {s1,s2} ->
                   {letrec smerge =  ... -- STGL translation
                    in let oute = {smerge,s1,s2} \u {} -> smerge {s1,s2}
                       in  sendChan {oute} }
     in letrec unshuffle =  ...  -- STGL-translation
        in letrec sort
             = {sendChan,merger,unshuffle,sort} \n {xs} ->
               case xs of
                 ...            -- case cascade
                 default  ->
                  case unshuffle {xs} of
                   {(l1,l2) ->
                     let inp1 = {sendChan,l1} \u {} -> sendChan {l1}
                         inp2 = {sendChan,l2} \u {} -> sendChan {l2}
                     in let {l1'} =  sortNet # {inp1}
                            {l2'} =  sortNet # {inp2}
                     in let sl1 = {sendChan,l1'} \u {} -> sendChan {l1'}
                            sl2 = {sendChan,l2'} \u {} -> sendChan {l2'}
                     in let {res} = merger # {sl1,sl2}
                     in res   }     }    }
        in let sorted = {sort,list} \u {} -> sort {list}
        in sendChan {sorted} }
```

◁

Communication using channel structures. As has been discussed in the context of the operational semantics before, the handling of channel structures requires information about types and annotations. For a given data type, the following recursive data type suffices to encode the way it is annotated:

```
data SendMode = Single | Comps [SendMode]
```

`Single` means that the present data type is transmitted using one channel and `Comps` represents the case that it is a channel structure. The list elements following this constructor represent the annotation status of the *type variables* involved.

We will introduce a new transmission function `sendStruct` that takes as its first argument the above mode information and the data to be transmitted as its second. In order to be able to use `sendStruct` for various data types, we introduce type class `Transmissible`, which contains operation `sendStruct` and additionally requires normal form evaluation, cf. class `NFData`[THLP98]. 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 types to be handled in this way can be determined at compile time. In practical programs this set 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. For them, they need to be instances of `Transmissible` with specific methods `sendStruct`.

For channel structures, `sendStruct` will perform pattern matching on the data structure to be sent in order to find out the respective constructor and arity.

It will then call the primitive function `split` which decomposes a channel structure into a data structure with channels and/or channel structures as its components. This function has as arguments the data constructor itself (which has to be suitably encoded in order to avoid type problems), its arity and an expression enclosed in {} that defines the new threads.

Example 8.2

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)

searchEng :: (Tree inf) -> Process  [req]       (Tree <[ans]>)
searchEng 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]
```

In `handle` further subprocesses can be spawned, if necessary. 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. For the programmer, this approach is convenient and safe, as the communication structure need not be flattened into lists.

For this binary tree `sendStruct` looks as follows:

```
sendStruct :: SendMode -> Tree a -> ()
sendStruct Single        x            = sendChan x
sendStruct (Comps [ann]) (Leaf x)     = split "Leaf" 1  {sendStruct ann x}
sendStruct (Comps [ann]) (Node x l r)
  = split "Node" 3
    {sendStruct ann x || sendStruct Comps [ann] l || sendStruct Comps [ann] r}
```

Consequently, the annotated type `Tree [<a>]` will be assigned the send mode `Comps [Comps [Single]]`. In the general case of a type with k type variables, the result would be `Comps` followed by list with k elements. This transformation can be performed by a function `toMode` which operates on the normalized representation of the outputs of a process instantiation. ◁

Above we have demonstrated the handling of channel structures. In particular, we have shown that `sendStruct Single` reduces to `sendChan`. As channel structures form a special case that will not be used very frequently, we will for simplicity work with `sendChan` directly in the following.

In the next section we will present the operational semantics of PEARL, which illustrates on an intermediate level the execution of Eden programs in a distributed machine.

8.3 DREAM

A parallel DREAM computation is performed by a system of extended STGM instances, so-called *Dreams*, each of which represents an Eden process. Internally, a Dream maintains multiple flows of control which implement the threads producing independent outputs in Eden processes. Every machine instance uses its own local heap which contains local data structures as well as references to buffers for receiving inputs. Special `queueMe` closures are used to suspend threads that try to read inputs which are not yet available. This way of thread synchronisation is common in distributed abstract machines. DREAM differs substantially from other abstract machines used in the implementation of parallel functional languages. The details of these differences are discussed in Chapter 8.7.

The set of the *Dreams*' states is called *DreamState*. The state of the whole DREAM is represented by a finite mapping of the set *Process_Id* of process numbers to *DreamState*. The transitions are denoted by $\Rightarrow$ and define a binary relation on the set of DREAM states:

$$\text{DREAM} = \langle \mathit{Process\_Id} \rightarrow_{\text{finite}} \mathit{DreamState},\ \Rightarrow \rangle,$$

where $\mathit{Process\_Id} \stackrel{\text{def}}{=} \mathbb{N}$. A detailed specification and explanation of the STGM can be found in [Pey92]. The sequential transitions of the STGM remain unchanged in the parallel context. They only have to be adapted to the extended state. In the following, we will show what has to be added to the STGM in order to express the full range of parallel computations that can be programmed in Eden.

8.3.1 Extension of the STG machine

Within a Dream, multithreading is used to concurrently compute the different outputs of the corresponding process. There is a one to one correspondence between machine instances and processes and between threads and outports.

See below how the state of one process in the abstract machine DREAM can be shown in the form of a table.

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) originate from the STG[Pey92], which also contains *heap* and global environment σ. The components printed in ***bold math*** have been added for the implementation of Eden.

More formally, the state of a Dream contains for each thread the following components, where *Int* is a set of tagged integer values, *Addr* is a set of tagged heap addresses, $Val := Int \cup Addr$, and $Env := [Var \rightarrow_{\text{finite}} Val]$ is the set of environments:

code	c	$\in Instr$, where *Instr* is defined below,
argument stack	as	$\in Val^*$,
return stack	rs	$\in Cont^*$, where $Cont = Alts \times Env$
update stack	us	$\in Frame$, where $Frame = Val^* \times Cont^* \times Addr$
outport (list)	$\boldsymbol{out}$	$\in (Process\_Id \times Channel\_Id)^*$

and, in common for all threads:

heap	h	$: Addr \rightarrow Closures$, where $Closures = Lfs \times Val^*$,
global environment	σ	$: Var \rightarrow Addr$,
inport table	$\boldsymbol{ipt}$	$: Channel\_Id \rightarrow Addr \times Process\_Id$,
counter of blocked threads	$\boldsymbol{b}$	$\in Int$.

The first four components of a thread state as well as the heap and the global environment exactly correspond to those of the STGM state except that we extend the underlying sets of expressions (PEARL expressions), lambda forms and closures (see below).

The *outport* component within a thread state contains information about the connection to an inport of another process instance which will be fed by the thread. An outport specification is a pair (m, c) where $m \in Process\_Id$ is the identifier of a remote Dream and $c \in Channel\_Id$ is a channel identifier in this remote machine. It denotes the destination of the output produced by the thread. In general each thread contains exactly one outport specification. There are two exceptions: The very first thread of a process which evaluates the process body before spawning the threads contains the *list of all outports* communicating with the parent process. The thread evaluating the main expression has *no* outport.

The *inport table* ***ipt*** maps channel identifiers to the respective heap addresses, where received messages are stored. Additionally, the sender's id is logged here to be able to check the 1:1 correspondence of inports to outports. The heap addresses point to `queueMe` closures which represent not yet received messages.

The *counter of blocked threads* is introduced to detect the termination of processes. A process terminates when the set of active threads is empty and there are no blocked threads in the heap.
The instructions are the same as those of the STGM:

$$\begin{array}{rcllll} Instr & = & \{Eval & e\ \rho & \mid e \in Expr, \rho \in Env\} \\ & \cup & \{Enter & a & \mid a \in Addr\} \\ & \cup & \{ReturnCon & (C\ w_1 \ldots w_n) & \mid C \in \Gamma, w_i \in Val\} \\ & \cup & \{ReturnInt & k & \mid k \in Int\} \end{array}$$

with the following intuitive meaning: *Eval* e ρ evaluates expression e in the environment ρ and applies its value to the arguments on the argument stack. *Enter* a applies the closure at address a to the arguments on the argument stack. *ReturnCon* c ws returns the constructor c applied to values ws to the continuation on the return stack. *ReturnInt* k returns the primitive integer k to the continuation on the return stack.

The set of heap closures is extended by a closure `queueMe` of the form:

$$\{\}\ \backslash n\ \{\}\texttt{->}\ \texttt{queueMe}\ q$$

which causes the suspension of a thread when entered, appending its state (code, argument stack, return stack, update stack, outport) to the associated queue q. In the following, we will represent this closure as a pair $\langle \texttt{queueMe}, [t_1, \ldots, t_n] \rangle$ where the t_i are thread states. This special closure is used to mark for stream channels the write end where new incoming values must be appended and for a one-value channel the position for storing the value. Moreover, it is used to prevent the simultaneous evaluation of updatable closures by multiple threads.

We introduce `ChanName` closures $\{\}\ \backslash n\ \{\}\texttt{->}\ \texttt{ChanName}\ (pid, cid)$ to represent the names of dynamic channels, where (pid, cid) is an outport specification.

The initial state of the system consists of a unique Dream with process id 0 embodying a thread for evaluating the main expression. Its local heap h_0 contains a closure for each globally defined variable, and the global environment σ_0 binds each global variable to the heap address of its closure. The input channel table is empty and there are no blocked threads. The initial thread has empty stacks denoted by ε and no outport specification.

$$\boxed{\langle 0 \mapsto ((Eval\ \ \textbf{main}\ \{\ \}\ \rho_\emptyset)\ \ \varepsilon\ \ \varepsilon\ \ \varepsilon\ \ \varepsilon)\ \ h_0\ \ \sigma_0\ \ [\,]\ \ 0\rangle}$$

8.3.2 DREAM transitions

The representation of a transition. The execution of the code for a thread may cause changes not only to the thread itself, but also to the whole process, i.e. the shared components of the corresponding Dream. Moreover, there are evaluations which cause interactions with other Dreams, thus affecting the state of the whole system; e.g. process instantiation or communication. In the specifications we adopt the notation used in [Pey92]. We only show the states of the machine instances involved in the transitions and use the symbol $\|$ to separate concurrent threads and $|||$ to separate parallel Dream instances. The length of a sequence xs is denoted by $|xs|$.

Alternatively, rules can be visualized in the form of diagrams that show the state of the process(es) before the transition in the top half and the ones after the transition in

the bottom half, separated by an arrow that illustrates the transition. Figure 8.2 gives an example of such a diagram for communication.

We will present the transitions in the following order: We start with communication, which is performed between threads of different instances, afterwards explain the interaction of threads in the same instance and finally explain process management.

Communication:

In PEARL, data is communicated using the primitive functions **sendHead**, **closeStrm** and **sendVal**.

Sending and receiving a single value: The outport in a thread's state contains a remote inport address consisting of a process instance number and a channel identifier. The remote inport table maps the channel identifier to a `queueMe`-closure address and a sender id, which in the case of a dynamically created channel will initially be unknown ($\perp$). The suspended threads collected in the `queueMe`-closure are reactivated, the counter of blocked threads is decremented accordingly, and the closure is updated with the transmitted closure using the auxiliary function *graph_copy*. In fact, the whole subheap which can be referenced by the sent closure must be transmitted and embedded into the receiver's heap.

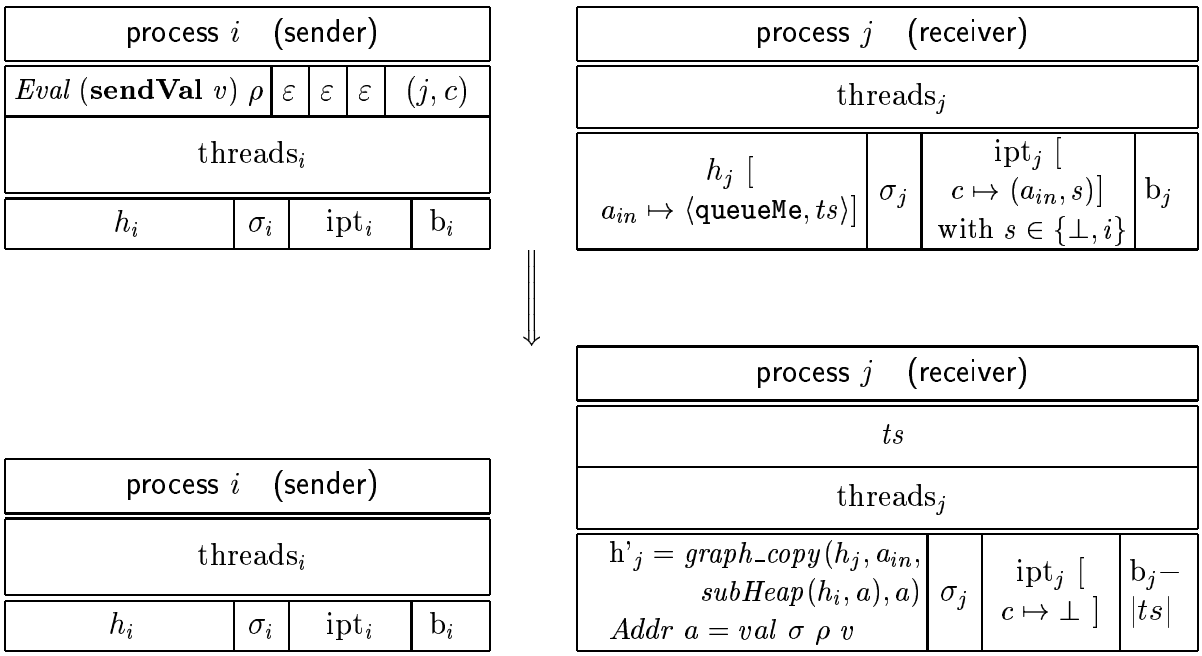


Figure 8.2: Visualization of communication transition (C.1)

Figure 8.2 shows the most straightforward transition in communication, namely C.1. The sending thread terminates after it has carried out the evaluation of **sendVal**, because its task was to carry out precisely this send operation. This output operation belongs to a one-value channel. In the diagram, the relation between the state of the communicating processes before and after the transition is indicated by a horizontal arrow.

In contrast to the above, for an evaluation of **sendHead**, which is performed for stream channels, the sending thread would have remained. Naturally, for one-value communica-

tion, the channel's id is removed from the receiver's inport table after the transition and for stream communication it is modified. These distinctions can be compared in detail in the specifications of rules (C.1) and (C.2) below. In order to save space, we will subsequently present all transitions in this "mathematical style" and not in the form of the above diagram.

$$
\begin{array}{l}
\langle i \mapsto ((Eval\ (\mathbf{sendVal}\ v)\ \rho)\ \varepsilon\ \varepsilon\ \varepsilon\ (j,c)) \parallel threads_i\ h_i\ \sigma_i\ ipt_i\ b_i\rangle \\
|||\quad \langle j \mapsto threads_j\ h_j[a_{in} \longmapsto \langle \texttt{queueMe}, ts\rangle]\ \sigma_j\ \ ipt_j[c \longmapsto (a_{in}, s)]\ b_j\rangle \\
\qquad \text{with } s \in \{\bot, i\} \\
\Rightarrow \\
\langle i \mapsto threads_i\ \ h_i\ \ \sigma_i\ \ ipt_i\ \ b_i\rangle \\
|||\quad \langle j \mapsto (ts \parallel threads_j)\ \ h'_j\ \ \sigma_j\ \ ipt_j[c \longmapsto \bot]\ \ b_j - |ts|\rangle \\
\qquad \text{where } h'_j = graph\_copy(h_j, a_{in}, subHeap(h_i, a), a) \\
\qquad\qquad\quad Addr\ a = val\ \sigma\ \rho\ v
\end{array}
\tag{C.1}
$$

$graph\_copy(h_1, a_1, h_2, a_2)$ creates a copy of heap h_2 with new unique addresses within another heap h_1 starting at address a_1 where the node $h_2(a_2)$ is inserted. To determine the subheap of a heap h which contains all nodes reachable from a list of node addresses ws we use the function *subHeap*. Details are omitted.

The sending thread terminates after completing its task. The receiving process adds the value to its heap and deletes the inport from the inport table.

Sending and receiving a stream element: Sending a stream of values is similar to sending a single value except that the sending of values and the termination of the thread are done by separate primitive functions.

$$
\begin{array}{l}
\langle i \mapsto ((Eval\ (\mathbf{sendHead}\ v)\ \rho)\ \varepsilon\ (default \to cont, \tilde{\rho}) : rs\ \varepsilon\ (j,c) \\
\qquad \parallel threads_i\ h_i\ \sigma_i\ ipt_i\ b_i\rangle \\
|||\quad \langle j \mapsto threads_j\ h_j[a_{in} \longmapsto \langle \texttt{queueMe}, ts\rangle]\ \sigma_j\ \ ipt_j[c \longmapsto (a_{in}, s)]\ b_j\rangle \\
\qquad \text{with } s \in \{\bot, i\} \\
\Rightarrow \\
\langle i \mapsto ((Eval\ cont\ \tilde{\rho})\ \varepsilon\ rs\ \varepsilon\ (j,c)) \parallel threads_i\ \ h_i\ \ \sigma_i\ \ ipt_i\ \ b_i\rangle \\
|||\quad \langle j \mapsto (ts \parallel threads_j)\ \ h'_j\ \ \sigma_j\ \ ipt_j[c \longmapsto (a_t, i)]\ \ b_j - |ts|\rangle \\
\qquad \text{where} \\
\qquad h'_j = graph\_copy(h_j[a_{in} \mapsto \langle \texttt{Cons}\ \{a_v, a_t\}\rangle,\ a_t \mapsto \langle \texttt{queueMe}, [\,]\rangle], \\
\qquad\qquad\qquad\qquad a_v, subHeap(h_i, a), a) \\
\qquad Addr\ a = val\ \sigma\ \rho\ v \\
\qquad a_v \text{ and } a_t \text{ are new heap addresses}
\end{array}
\tag{C.2}
$$

Streams are stored in the receiving process heap as normal lists, the only difference being that a `queueMe`-closure marks the end of the sequence of values received up to now. This closure will be rewritten when a new message for this stream arrives. Thus, sending a stream element updates the inport's closure with a new closure representing a list, the head of which points to the transmitted subheap and the tail of which is a new `queueMe`-closure with an empty list of suspended threads. The inport table is updated with the address of the new `queueMe`-closure. The sending thread continues its computation with the continuation given on top of the return stack. In fact, the continuation is the evaluation and sending of the remaining list via the channel.

Closing a channel: The closing of a channel implies the termination of the corresponding thread. On the receiver's side the threads depending on the inport are reactivated, and the `queueMe`-closure is overwritten with a `Nil` closure marking the end of the list associated to the stream.

$$
\begin{array}{l}
\langle i \mapsto ((Eval\ \ \textbf{closeStrm}\ \ \rho)\ \ \varepsilon\ \ \varepsilon\ \ \varepsilon\ \ (j,c)) \parallel threads_i\ \ h_i\ \ \sigma_i\ \ ipt_i\ \ b_i\rangle \\
|||\ \langle j \mapsto threads_j\ \ h_j[a_{in} \longmapsto \langle \texttt{queueMe}, ts\rangle]\ \sigma_j\ \ ipt_j[c \longmapsto (a_{in}, s)]\ \ b_j\rangle \\
\Rightarrow \\
\langle i \mapsto threads_i\ \ h_i\ \ \sigma_i\ \ ipt_i\ b_i\rangle \\
|||\ \langle j \mapsto (ts \parallel threads_j)\ \ h_j[a_{in} \longmapsto \langle \texttt{Nil}\rangle]\ \sigma_j\ \ ipt_j[c \longmapsto \bot]\ \ b_j - |ts|\rangle
\end{array}
\tag{C.3}
$$

Receiving from a channel structure When the sender evaluates the output associated with channel structure c to a term $C\ v_1 \ldots v_n$, this channel structure will be split up into the components $c_1 \ldots c_n$, connected by the same n-ary constructor C. Generally, this set of components can contain channels and channel structures. The information about the number of components and their type is known to the specific method `sendStruct`. Apart from that, the handling of channels and channel structures is identical.

$$
\begin{array}{ll}
\langle i \mapsto & ((Eval\ (\textbf{split}\ C\ n\{\texttt{sendStruct}\ s_1\ v_1\ ||\ldots||\ \texttt{sendStruct}\ s_n\ v_n)\ \rho)\ \varepsilon\ \varepsilon\ \varepsilon\ (j,c)) \\
 & \parallel threads_i\ h_i\ \sigma_i\ ipt_i\ b_i\rangle \\
||| & \langle j \mapsto threads_j\ h_j[a \longmapsto \langle \texttt{queueMe}, ts\rangle]\ \sigma_j\ \ ipt_j[c \longmapsto (a,i)]\ b_j\rangle \\
\Rightarrow & \\
\langle i \mapsto & ((Eval\ \ (\texttt{sendStruct}\ s_1\ v_1)\ \rho)\ \varepsilon\ \varepsilon\ \varepsilon\ (j,c_1)) \\
 & \parallel \ldots \parallel \\
 & ((Eval\ \ (\texttt{sendStruct}\ s_n\ v_n)\ \rho)\ \varepsilon\ \varepsilon\ \varepsilon\ (j,c_n)) \\
 & \parallel threads_i\ \ h_i\ \ \sigma_i\ \ ipt_i\ \ b_i\rangle \\
||| & \langle j \mapsto (ts \parallel threads_j)\ \ h_j[\ a \longmapsto \langle C\{a_1,\ldots,a_n\}\rangle, \\
 & \qquad a_1 \longmapsto \langle \texttt{queueMe}, [\,]\rangle, \ldots, \\
 & \qquad a_n \longmapsto \langle \texttt{queueMe}, [\,]\rangle] \\
 & \sigma_j\ \ ipt_j[c \longmapsto \bot,\ c_1 \longmapsto (a_1,i), \ldots, c_n \longmapsto (a_n,i)]\ \ b_j - |ts|\rangle
\end{array}
\tag{C.4}
$$

When the receiver tries to access the respective inport, it will react to this change by inserting the respective term information at the address of the old inport and by introducing the n new inports. The 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 structure transmitted over a single regular channel and one transmitted using a channel structure.

Dynamic reply channels: The creation of a dynamic channel means the creation of a new inport, i.e. the inport table is extended by an entry which contains the address of a newly allocated `queueMe`-closure. As the sender to this inport is not yet known, $\bot$ is noted as sender id. When the first value of a stream is sent, $\bot$ will be overwritten by the

actual sender's id. In addition a `ChanName`-closure is included in the heap which contains the machine instance number and the new channel identifier.

$$
\begin{array}{l}
\langle i \mapsto ((Eval\ \mathbf{new}\ (chn, ch)\ e\ \rho)\ \ as\ \ rs\ \ us\ \ outp) \parallel threads\ \ h\ \sigma\ ipt\ b\rangle \\
\Rightarrow \\
\langle i \mapsto ((Eval\ \ e\ \ \rho[chn \mapsto Addr\ achn, ch \mapsto Addr\ ach])\ as\ rs\ us\ outp) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \parallel threads\ h'\ \ \sigma\ \ ipt'\ b\rangle \\
\qquad \text{where } ipt' = ipt[c \longmapsto (ach, \bot)] \\
\qquad\qquad\quad h' = h[achn \longmapsto \langle \texttt{ChanName}\ \ (i, c)\rangle, ach \longmapsto \langle \texttt{queueMe}, [\,]\rangle] \\
\qquad\qquad\quad achn \text{ and } ach \text{ are new heap addresses,} \\
\qquad\qquad\quad c \text{ is a new channel identifier}
\end{array}
\tag{D.1}
$$

When using a dynamic channel the current thread is split into two: one will continue with the evaluation of the main expression, and the other will be responsible for the new output.

$$
\begin{array}{l}
\langle i \mapsto ((Eval\ \ v\,!{*}e_1\ \mathbf{par}\ e_2\ \rho)\ as\ rs\ us\ outp) \parallel threads\ h\ \sigma\ ipt\ b\rangle \\
\qquad \text{with } val\ \ \rho\ \ \sigma\ \ v = Addr\ \ a \text{ and } \ h(a) = \langle \texttt{ChanName}\ \ out_{\text{dyn}}\rangle \\
\Rightarrow \\
\langle i \mapsto ((Eval\ e_1\ \rho)\ \varepsilon\ \varepsilon\ \varepsilon\ \ out_{\text{dyn}}) \parallel \\
\qquad ((Eval\ e_2\ \rho)\ as\ rs\ us\ outp) \parallel threads\ \ h\ \sigma\ ipt\ b\rangle
\end{array}
\tag{D.2}
$$

Organizing multithreading

Entering an updatable closure: In order to prevent multiple threads from simultaneously evaluating the same closure, an updatable closure is overwritten with a `queueMe`-closure on first entering it.

$$
\begin{array}{l}
\langle i \mapsto ((Enter\ a)\ as\ rs\ us\ outp) \parallel threads\ h\ \sigma\ ipt\ b\rangle \\
\qquad \text{with } h(a) = \langle (vs \setminus u\ \{\}\ \rightarrow\ e)\ ws_f\rangle \\
\Rightarrow \\
\langle i \mapsto ((Eval\ e\ [vs/ws_f])\ \varepsilon\ \varepsilon\ (as, rs, a) : us\ outp) \parallel threads\ h'\ \sigma\ ipt\ b\rangle \\
\qquad \text{where } h' = h[a \longmapsto \langle \texttt{queueMe}, [\,]\rangle]
\end{array}
\tag{M.1}
$$

Updating a closure: When the `queueMe`-closure is finally updated, the suspended threads are reactivated. This is the only difference between the following rules and the corresponding sequential ones. Updates are triggered in two cases. Firstly, when a *ReturnCon* instruction is executed with an empty return stack. Secondly, when a closure is entered for which there are not enough addresses on the argument stack (partial application).

$$
\begin{array}{l}
\langle i \mapsto ((ReturnCon\ c\ ws)\ \varepsilon\ \varepsilon\ (as_u, rs_u, a_u) : us\ outp) \parallel ts\ h\ \sigma\ ipt\ b\rangle \\
\qquad \text{with } h(a_u) = \langle \texttt{queueMe}, bts\rangle \\
\Rightarrow \\
\langle i \mapsto ((ReturnCon\ c\ ws)\ as_u\ rs_u\ us\ outp) \parallel ts \parallel bts\ h'\ \sigma\ ipt\ b{-}|ts|\rangle \\
\qquad \text{where } h' = h[a_u \longmapsto \langle vs \setminus n\ \{\} \rightarrow\ c\ vs)\ ws\rangle] \\
\qquad\qquad\quad vs \text{ is a list of distinct variables with } |vs| = |ws|
\end{array}
\tag{M.2}
$$

$$
\begin{array}{l}
\langle i \mapsto ((Enter\ a)\ \ as\ \ \varepsilon\ \ (as_u, rs_u, a_u) : us\ \ outp) \parallel threads\ \ h\ \ \sigma\ \ ipt\ b\rangle \\
\quad \text{with } h(a_u) = \langle \texttt{queueMe}, ts\rangle \\
\qquad\quad h(a) = \langle (vs \backslash n\ xs \to e)\ \ ws\rangle \\
\qquad\quad |as| < |xs| \\
\Rightarrow \\
\langle i \mapsto ((Enter\ a)\ as{+\!\!+}as_u\ rs_u\ us\ outp) \parallel ts \parallel threads\ h'\ \sigma\ ipt\ b{-}|ts|\rangle \\
\quad \text{where } xs_1 +\!\!+ xs_2 = xs \text{ with } |xs_1| = |as| \\
\qquad\quad f \text{ is an arbitrary variable} \\
\qquad\quad h' = h[a_u \longmapsto \langle ((f : xs_1) \backslash n\{\ \} \to f\ xs_1)\ \ a : as\rangle]
\end{array}
\tag{M.3}
$$

Suspending a thread: When a thread enters a `queueMe`-closure, its state is saved within this closure, and the number of blocked threads is incremented.

$$
\begin{array}{l}
\langle i \mapsto ((Enter\ a)\ as\ rs\ us\ outp) \parallel threads\ h\ \sigma\ ipt\ b\rangle \\
\quad \text{with } h(a) = \langle \texttt{queueMe}, ts\rangle \\
\Rightarrow \\
\langle i \mapsto threads\ h'\ \sigma\ ipt\ (b+1)\rangle \\
\quad \text{where } h' = h[a \longmapsto \langle \texttt{queueMe}, ((Enter\ a)\ as\ rs\ us\ outp) \parallel ts\rangle]
\end{array}
\tag{M.4}
$$

Process instantiation:

Now that we have presented the mechanisms used for message passing and for the interaction of threads in the same process, we can explain the creation of processes, which relies on both. Process instantiations only appear as special bindings in local declarations, i.e. let and letrec expressions. Thus, the rule for evaluating these expressions must be extended:

$$
\begin{array}{l}
\langle i \mapsto\ ((Eval\ \textbf{let}(\textbf{rec})\ binds\ \textbf{in}\ e\ \rho)\ as\ rs\ us\ outp) \parallel threads\ h\ \sigma\ ipt\ b\rangle \\
\Rightarrow \\
\langle i \mapsto ((Eval\ e\ \rho')\ as\ rs\ us\ outp) \parallel threads \parallel ts\ h'\ \sigma\ ipt'\ b\rangle \\
|||\ newInsts \\
\quad \text{where } \rho' = extend\_env(\rho, binds) \\
\qquad\quad (ts, h', ipt', newInsts) = handle\_bindings(i, binds, \rho', h, \sigma, ipt)
\end{array}
\tag{P.1}
$$

The function *extend_env* allocates space for closures and maps the variables on the left hand side of the bindings to the new heap addresses:

$$
\begin{array}{l}
extend\_env(\rho, \{\}) = \rho \\
extend\_env(\rho, \{x = lf; bind_2; \ldots; bind_n\}) \\
\quad = extend\_env(\rho[x \mapsto Addr\ a], \{bind_2; \ldots; bind_n\}) \\
\qquad \text{where } a \text{ is a new heap address} \\
extend\_env(\rho, \{\{o_1, \ldots, o_m\} = p\#\{i_1 \parallel \ldots \parallel i_k\}; bind_2; \ldots; bind_n\}) \\
\quad = extend\_env(\rho[o_1 \mapsto Addr\ a_1, \ldots, o_m \mapsto Addr\ a_m], \{bind_2; \ldots; bind_n\}) \\
\qquad \text{where } a_1, \ldots, a_m \text{ are new heap addresses}
\end{array}
$$

The corresponding heap closures are produced by *handle_bindings*. If the bindings contain process instantiations, this function also produces new threads, further entries in the inport table and new machine instances.

$$\begin{aligned}
&\mathit{handle_bindings}(i, \{\mathit{bind}_1; \ldots; \mathit{bind}_n\}, \rho, h_0, \sigma, \mathit{ipt}_0)\\
&\quad = (\mathit{threads}_1 \parallel \ldots \parallel \mathit{threads}_n, h_n, \mathit{ipt}_n, \mathit{newInst}_1 \mid\mid\mid \ldots \mid\mid\mid \mathit{newInst}_n)\\
&\quad \text{where for } 1 \leq j \leq n:\\
&\qquad\qquad (\mathit{threads}_j, h_j, \mathit{ipt}_j, \mathit{newInst}_j) = \mathit{handle_binding}(i, \mathit{bind}_j, \rho, h_{j-1}, \sigma, \mathit{ipt}_{j-1})
\end{aligned}$$

Rule (P.1) specifies that the current thread continues with the evaluation of the body expression e in the extended environment. The heap and inport table are adjusted accordingly. The new threads ts run concurrently and the newly generated machine instances *newInsts* run in parallel.

The auxiliary function *handle_binding* distinguishes between two cases. The simple case of binding is the heap allocation of a closure as in the sequential STGM. Only the heap is modified. In this case the effect of the transition exactly corresponds to the sequential rule. Let $\mathit{val}\ \rho\ \sigma\ x = \mathit{Addr}\ a$. Then:

$$\begin{aligned}
&\mathit{handle_binding}(i, x = \mathit{vars}_f \backslash\pi\ \mathit{vars}_a \rightarrow e_{\text{rhs}}, \rho, h, \sigma, \mathit{ipt}) = (\varepsilon, h', \mathit{ipt}, \varepsilon)\\
&\quad \text{where } h' = h[a \longmapsto \langle(\mathit{vars}_f \backslash\pi\ \mathit{vars}_a \rightarrow e_{\text{rhs}})\ (\rho\ \mathit{vars}_f)\rangle]
\end{aligned}$$

A process instantiation leads to the creation of new input channels, new threads for the evaluation of outports and a new machine instance for the newly created process. Let $\mathit{val}\ \rho\ \sigma\ p = \mathit{Addr}\ a$ and $h(a) = \langle(xs\ \backslash\mathbf{process}\ is \rightarrow e_{\text{body}})\ ws\rangle$. Then:

$$\begin{aligned}
&\mathit{handle_binding}(i, os = p\#vs, \rho, h, \sigma, \mathit{ipt}) = (\mathit{newThs}, h', \mathit{ipt}', \mathit{newInst})\\
&\quad \text{where } (\mathit{new}_i, \mathit{newInst}, \mathit{outs})\\
&\qquad\qquad = \mathit{create_process}(\langle(xs\ \backslash\mathbf{process}\ is \rightarrow e_{\text{body}})\ ws\rangle, h_{ws}, i, \mathit{ins}, \sigma)\\
&\qquad (\mathit{ins}, h', \mathit{ipt}') = \mathit{create_inports}(i, \mathit{new}_i, os, \rho, h, \mathit{ipt})\\
&\qquad h_{ws} = h_0 \cup \mathit{subHeap}(h, ws)\\
&\qquad \mathit{newThs} = \mathit{spawn_threads}(vs, \rho, \mathit{outs})
\end{aligned}$$

In the state of the current machine instance, input channels are created by *create_inports* to receive messages from the outports of the new process. The identifiers of these channels together with the parent id are passed as arguments to the function *create_process* that yields a new machine instance for the process to be created. The function *spawn_threads* creates for each inport of the newly created process a new thread which evaluates the associated expression.

The heap h_{ws} contains the static closures of the global environment in h_0 and all the relevant closures needed to correctly bind the free variables xs of the process abstraction as well as all closures reachable from these closures via variable bindings computed by *subHeap*[3]. It is used as the initial heap for the new machine instance. The auxiliary functions *create_inports*, *create_process* and *spawn_threads* are defined next.

create_inports creates new input channels. It allocates `queueMe`-closures in the heap and extends the environment by mapping the input variables in_i to the addresses of the new closures. Finally, it extends the inport table by mappings of new channel identifiers to the `queueMe`-closures. The updated state components and the list of the new channel names are returned as result. Let $\mathit{val}\ \rho\ \sigma\ in_j = \mathit{Addr}\ ai_j$ for $1 \leq j \leq n$. Then:

[3]Note that this heap must not contain any `queueMe`-closure. Otherwise the process instantiation will be suspended until this closure is overwritten.

$$
\begin{aligned}
&\mathit{create\_inports}(i_{\text{recv}}, i_{\text{send}}, [in_1, \ldots, in_n], \rho, h, ipt) \\
&\quad = ([(i_{\text{recv}}, c_1), \ldots, (i_{\text{recv}}, c_n)], h', ipt') \\
&\quad \text{where } h' = h[ai_1 \longmapsto \langle \texttt{queueMe}, [\,]\rangle, \ldots, ai_n \longmapsto \langle \texttt{queueMe}, [\,]\rangle] \\
&\qquad\quad ipt' = ipt[c_1 \mapsto (ai_1, i_{\text{send}}), \ldots, c_n \mapsto (ai_n, i_{\text{send}})] \\
&\qquad\quad c_1, \ldots, c_n \text{ are new channel identifiers}
\end{aligned}
$$

m *create_process* is used to define a new machine instance with a new identifier and a list of outport specifications. The state of this new machine contains the threads for its outports, the heap passed as a parameter extended by the inports of the process, the global environment and the new inport table. The threads for the outports are spawned using *spawn_threads*.

The addresses of the corresponding inports (of the parent process) are passed as a parameter. Inport channels are created by *create_inports* to which the new outports of the parent process will be connected.

$$
\begin{aligned}
&\mathit{create\_process}(\langle (xs \setminus \textbf{process } is \ \rightarrow \ e_{\text{body}})\ ws\rangle, h_{ws}, i, outs, \sigma) \\
&\quad = (new_i, [new_i \mapsto state_i], ins) \\
&\quad \text{where } \rho_0 = [xs \mapsto ws, i_1 \mapsto Addr\ a_1, \ldots, i_n \mapsto Addr\ a_n] \\
&\qquad\quad state_i = ((Eval\ e_{\text{body}}\ \rho_0)\ \varepsilon\ \varepsilon\ \varepsilon\ outs)\ h_{in}\ \sigma\ ipt_{in}\ 0 \\
&\qquad\quad (ins, h_{in}, ipt_{in}) = \mathit{create\_inports}(new_i, i, is, \rho_0, h_{ws}, [\,]) \\
&\qquad\quad new_i \text{ is a new process identifier,} \\
&\qquad\quad a_1, \ldots, a_n \text{ are new heap addresses.}
\end{aligned}
$$

spawn_threads takes a parallel expression of the form $\{e_1 \parallel \ldots \parallel e_k\}$, an environment ρ, and a list of outport specifications and generates threads. These evaluate the expressions in the given environment and send the results via the respective outports.

$$
\begin{aligned}
&\mathit{spawn\_threads}\ (\{e_1 \parallel \ldots \parallel e_k\}, \rho, [out_1, \ldots, out_k]) = t_1 \parallel \ldots \parallel t_k \\
&\quad \text{where } \ t_i = ((Eval\ e_i\ \rho)\ \varepsilon\ \varepsilon\ \varepsilon\ out_i),\ \forall i \in \{1, \ldots, k\}
\end{aligned}
$$

The evaluation of a process body leads to a parallel expression which defines the threads for the outports to the generator process. The single control thread for the creation of the heap and top level subprocesses terminates. k threads for the evaluation of the outports are started.

$$
\boxed{
\begin{aligned}
&\langle i \mapsto ((Eval\ \{e_1 \parallel \ldots \parallel e_k\}\ \rho)\ \ \varepsilon\ \varepsilon\ \varepsilon\ outs) \parallel threads\ h\ \sigma\ ipt\ b\rangle \\
&\Rightarrow \\
&\langle i \mapsto threads' \parallel threads\ h\ \sigma\ ipt\ b\rangle \\
&\qquad \text{where } threads' = \mathit{spawn\_threads}(\{e_1 \parallel \ldots \parallel e_k\}, \rho, outs)
\end{aligned}
}
\tag{P.2}
$$

Termination:

A machine instance without any threads (active or blocked) terminates immediately. The active or blocked threads in other instances which fill its inports are removed using the auxiliary function *remove_thread*. This function removes an active or blocked thread associated with an outport given as the first argument from a machine state given as the second argument (see below).

$$\begin{array}{l}\langle i \mapsto \emptyset\ h\ \sigma\ [c_1 \mapsto (j_1, a_1), \ldots, c_k \mapsto (j_k, a_k)]\ 0\rangle \\ |||_{l=1}^{k}\ \langle j_l \mapsto state_l\rangle \\ \Rightarrow \\ |||_{l=1}^{k}\ \langle j_l \mapsto remove\_thread\ (i, c_l)\ state_l\rangle\end{array} \tag{P.3}$$

where
$remove\_thread\ outp\ ((instr\ as\ rs\ us\ outp)\ \|\ threads\ h\ \sigma\ ipt\ b)$
$\quad = (threads\ h\ \sigma\ ipt\ b)$
$remove\_thread\ outp\ (threads\ h[a \mapsto \langle \texttt{queueMe}, (instr\ as\ rs\ us\ outp)\ \|\ threads'\rangle]\ \ \sigma\ ipt\ b)$
$\quad = (threads\ h[a \mapsto \langle \texttt{queueMe}, threads'\rangle]\ \sigma\ ipt\ (b-1))$

Nondeterminism:

The predefined `merge` process passes values from its incoming channels to its outgoing channel. We omit the straightforward definition of this primitive process.

Summary: This completes the specification of the DREAM machine. It adds six new transition rules for communication (C.1 – C.4, D.1, D.2), modifies the rules for entering an updatable closure (M.1) and for updating closures (M.2, M.3) and introduces a new rule for thread suspension (M.4), it extends the evaluation of *letrec*-expressions by process instantiations (P.1) and finally adds rules for thread spawning (P.2) and process termination (P.3). In summary, nine transition rules have been added and four rules have been modified in order to build up a parallel system on top of the sequential STGM [Pey92].

8.4 Implementing DREAM

In the following we will explain how the abstract machine DREAM can be used as a basis for a parallel runtime system. The *front end* of the GHC transforms Haskell programs into *Core Haskell*, a minimal functional language in which all other Haskell constructs can be expressed. The *back end* generates C-code for the evaluation of Core Haskell programs. C is used as a portable target language of the compiler. The resulting C program is finally compiled with the gcc for the respective target machine.

In the following we describe our extensions of these parts in order to compile Eden. We start with a bird's eye view on the GHC.

8.4.1 Starting point: the compiler for Haskell

In order to bridge the gap between functional languages and the target machine's native code the compilation is split up into several passes (as shown in Figure 8.3). Each pass transforms the program into a lower level intermediate language.

In the first passes, information about the source program is gathered. Functional languages are statically typed, but in programs type declarations are not mandatory. Hence, the types have to be inferred in a separate pass. The *type inference* is performed on the full Haskell syntax to be able to return useful error messages for the programmer.

The key idea of the GHC is "compilation by program transformation"[San95]: The Haskell programs are first simplified to Core Haskell programs. Core Haskell can then be easily transformed into a still functional abstract machine language STGL which has a

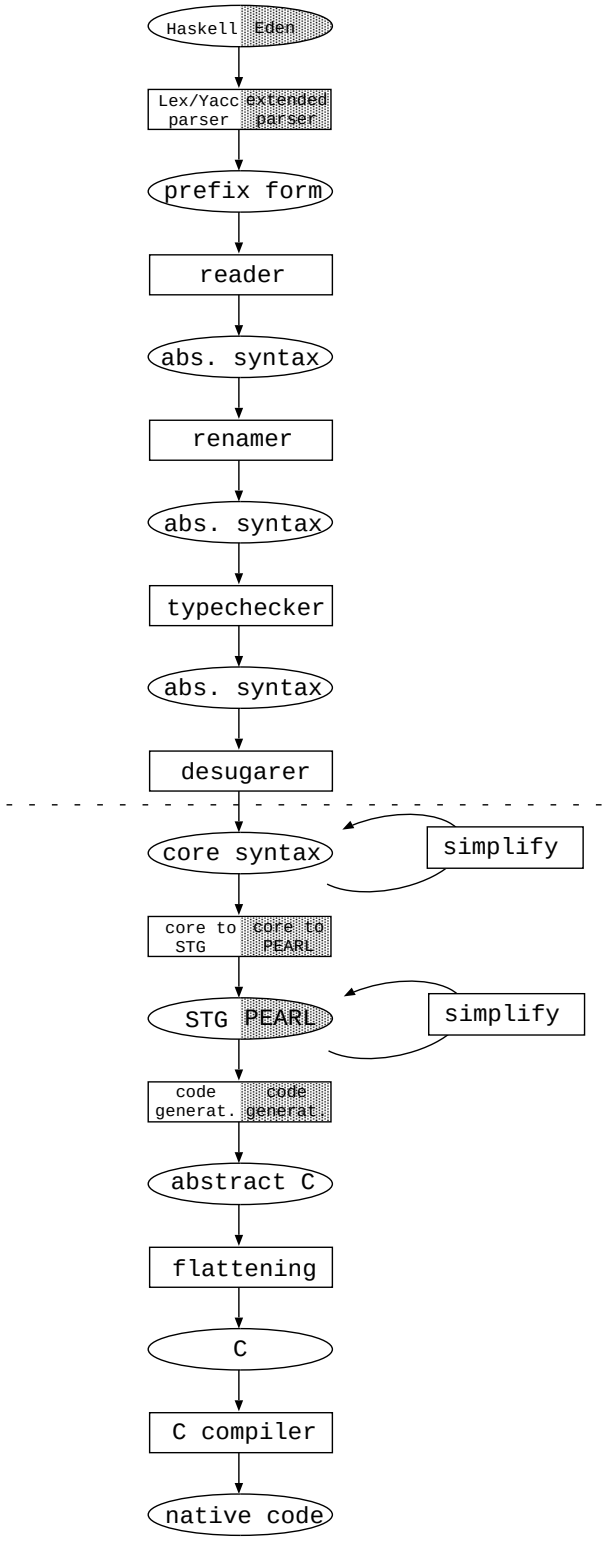


Figure 8.3: Overview of GHC's compiler passes and the extensions introduced to compile Eden (marked in grey)

simple denotational and operational semantics. The operational semantics is defined by the Spineless Tagless G machine (STGM) which is an abstract description of the run-time system [Pey92]. The compilation of STGL into C respects the operational semantics of the expressions.

8.4.2 Primitive extensions to the front end

From an implementation point of view, Eden's syntactic extensions to Haskell can be seen as "syntactic sugar" in the front end. We introduce for each parallel construct a *pseudo Haskell function* with the types chosen appropriately. For these functions only an interface with the functions' type declarations, but no implementation is provided.

We specify the Eden datatype `Process a b` to distinguish between processes and functions and introduce the pseudo functions:

```
process :: (a -> b) -> Process a b
(#) :: Process a b -> a -> b
```

A process abstraction of the form

```
process (i1, ..., in) (o1, ..., om)
where decls
```

is transformed into the expression:

```
process ( \ (i1, ..., in) -> let decls
                             in  (o1, ..., om) )
```

Process instantiations need not be changed. The type of the operator `(#)` for process instantiation ensures that only process abstractions are applied to input values, and that the number and types of inputs and outputs match. The other Eden constructs have been handled in the same manner.

To sum up, we have only modified the parser in order to embed Eden's additional expressions in Haskell syntax. The resulting Haskell program is passed through the subsequent, unchanged compiler passes, until Core Haskell syntax is reached. In particular, the original type inference algorithm is used to check the types of Eden programs.

8.4.3 Orthogonal extensions to the back end

In our intermediate language PEARL, multithreading and communication are expressed explicitly. The following syntax is used:

```
{expr || ... || expr}                        -- multithreading
{frees} \p {args} body                       -- process abstraction
{var, ..., var} = p # {expr || ... || expr} -- process instantiation
```

The multithreading expression means that the component expressions should be evaluated concurrently, each by a separate newly created thread. It is used in process instantiations and in the body of process abstractions to indicate the generation of threads. A process abstraction is represented by a lambda abstraction with free variables `frees`, flag `p` and parameters `args`.

Communication is made explicit by introducing the overloaded functions `sendChan` and `sendStruct` mentioned before to send values across communication channels. In each of their instances, these functions force the evaluation of the arguments and rely on the primitive functions **sendVal**, **sendHead**, **closeStrm**, and **split** to perform the communication. The primitive functions are implemented by routines of the underlying message passing system.

In order to retrieve PEARL expressions from the Core Haskell expressions with embedded Eden constructs, we have modified the compiler pass translating Core Haskell to STGL. In fact, we first translate Core Haskell into STGL and then gather the parts necessary for constructing PEARL expressions.

The following translation schemes introduce the PEARL expressions for process abstraction and instantiation. For better readability we show only the body of STG lambda forms without arguments.

```
STG syntax:   var = process lf
              lf  = { frees } \n { args } body
              lf :: t -> (t1, ..., tn)
-->
PEARL syntax: var    = { frees } \p { args } body'
              body'  = scheme body
              scheme = {} \n x -> case x of
                                    (x1, ..., xn) ->
                                      {sendChan x1 ||...|| sendChan xn}
```

The original process abstraction is transformed into a lambda abstraction with flag `p`. In order to introduce the multithreading expression and communication function `sendChan` in the body of the process abstraction, we extract the number of output channels from the type of the original lambda form and apply a general scheme for the process body to the actual body expression. In the GHC, the type information is still accessible in the STG program.

```
STG syntax:    var = (#) p arg
               arg = (vari_1, ..., vari_n)
               p :: Process (it_1, ..., it_n) (ot_1, ..., ot_m)
-->
PEARL syntax:  var = (varo_1,...,varo_m)
               {varo_1,...,varo_m}
                   = p # {sendChan vari_1 ||...|| sendChan vari_n}
```

In STGL, the pseudo function `(#)` indicates a process instantiation. The input tuple of this instantiation will be defined separately, as functions are applied to variables only. The type of the process abstraction is used to determine the number of outputs of the instantiation. PEARL's process instantiation provides variables for the inputs and outputs. The multithreaded evaluation and the communication of the inputs of the new process are made explicit. Of course, the output tuple of the process instantiation has to be reassigned to the original variable `var` as a tuple.

The most important part of the back end is of course the translation into C. The compilation of STGL expressions and declarations remain unchanged. The PEARL transformation is determined by the parallel run-time system which is discussed in the following.

8.5 Eden's Parallel Runtime System

In this section we give an outline of the parallel runtime system of Eden based on DREAM. We will show the advantages of the design decisions underlying Eden with respect to efficiency and ease of implementation in a distributed setting. Proceeding from an arbitrary implementation of the computation language we devise extensions for the following aspects:

1. management of parallel activities: representation of processes and threads, dynamic process management, multithreading inside of processes, scheduling

2. communication between processes.

In the following we will explain the specific requirements and underlying principles independently of the concrete sequential system used as basis. In Chapter 8.6 the details of the implementation are discussed.

8.5.1 Management of parallel activities

The representation of processes and threads:

An Eden process consists of one or more concurrent threads of control. These evaluate different output expressions which are independent of each other and use a common heap that contains shared information.

In the state of a process, the interface is visible in the following way: the outports are associated to threads (runnable or blocked) and the inports are enumerated in the inport table.

If a new output is introduced, e.g. due to the creation of a child process, the number of active threads increases. Threads can become blocked if they try to access information which is not yet available. Such threads will be enqueued and removed from the set of active threads. In this case, the counter for blocked threads will be incremented correspondingly.

Process management:

The parallel runtime system consists of a finite number of logical processor elements. Each processor element executes several Eden processes concurrently.

Process creation: The evaluation of a process instantiation leads to the creation of a child process which will be spawned on a processor element selected by a load balancing mechanism.

The complete subheap which represents the process abstraction, including the bindings of free variables, is sent to the assigned processor. In this way, process abstractions

are closed objects without any references to non-local data. The interface for the communication is installed for both processes by the generation of inport representations in the heap and the generation of threads for the computation of the outgoing values.

Process termination: In general, processes without any outports will be terminated as they cannot contribute to the overall result of the program. Note that the absence of active threads does not imply that there are no outports left. Blocked threads have to be taken into account. For the organization of the termination a backwards referencing scheme from inports to outports is necessary. Whenever a process terminates, it closes its inports and informs the connected outports that they should terminate as well.

Multithreading:

In order to prevent multiple threads of the same process from evaluating the same shared expression, a simple synchronisation mechanism is used. The thread that is the first to access the expression, starts its evaluation and overwrites the heap representation by a special object, called `queueMe`-closure, that indicates that the data is not yet available. Subsequent threads which attempt to access this object will be blocked in a special queue associated with it. All members of the queue will be released when the first thread overwrites the `queueMe`-closure with the result of the computation. The same mechanism can be found in most of the parallel runtime systems for functional languages.

Scheduling:

Eden needs a fair scheduler to guarantee safety, because the evaluations of concurrent threads are not directly demand driven. Every logical PE runs a scheduler, which has to:

- perform local garbage collection
- process incoming messages
- run the threads
- balance the speed of execution between senders and consumers, so that computation can proceed as fast as possible.

8.5.2 Communication

Communication channels are only represented by in- and outports which are connected to each other. The outport is just the global reference to an inport. These are the only global references needed. There are no explicit channel buffers, but the data is directly transferred from the producer's heap to the consumer's heap using the inport to find the location where the data should be stored.

In contrast to most other parallel systems based on lazy functional languages, message data is evaluated and transmitted eagerly in Eden. This means that a one message protocol is used for communication, i.e. one that dispenses with request messages for remote data. This feature is very important for the efficient implementation on distributed memory systems with considerable communication latencies, as it reduces the number of small control messages. It furthermore alleviates the overlapping of computation and

communication. The threads will be suspended less frequently than in a two message protocol, where a thread *must* be blocked after requesting the remote data. Nevertheless, message passing will not be done blindly, but mechanisms for controlling the flow of messages will be provided as refinements (see Chapter 8.6).

The sending of data across communication channels is initiated in special routines which correspond to the predefined communication functions in PEARL. The receipt of any messages is handled by the scheduler which puts the message contents into the heap and updates the inport table. The necessary messages and their handling will be described in the next section.

In Eden, all communication takes place via message passing. The interface of a process shows the communication partners currently connected to it. No information can be implicitly shared among different processors. Consequently, the communication requirements are clear and well-defined. This principle is closely related to the fact that messages contain only data in normal form (see above).

8.6 Implementation details

The implementation of Eden's parallel runtime system makes use of the Glasgow parallel Haskell system GUM, an extension of the GHC to support parallelism introduced by program annotations. The original GUM system uses PVM (Parallel Virtual Machine) to express parallelism and communication. We have ported the whole system to the MPI standard [Plü97].

In this section we describe how the GUM system is modified in order to support the evaluation of Eden programs. We start with a short description of GUM.

8.6.1 GUM — a parallel functional runtime system

In GUM the units of computation are called threads. Each *thread* evaluates an expression to weak head normal form. A thread is represented by a heap-allocated Thread State Object (TSO) containing slots for the thread's registers and a pointer to heap-allocated Stack Objects (SO). Each processor element (PE) has a pool of runnable threads. Due to the demand driven evaluation it is sufficient to use a non-preemptive thread scheduler where each thread is run until it completes, space is exhausted or the thread blocks. But the system also provides a fair scheduler which executes threads in following a round-robin policy with time slices. In order to avoid that two threads evaluate the same heap closure, the synchronisation mechanism explained in Chapter 8.5.1 is used. The `queueMe`-closures are called black holes. A thread that enters a black hole saves its state in its TSO and attaches the address of its TSO to the blocking queue associated with the black hole.

The memory management in GUM is especially involved as it has to manage a virtual shared memory in which the shared program/data graph resides. When a closure is moved from one PE to another, e.g. in the process of work distribution, the original closure is overwritten with a Fetch-Me closure which contains the global address of the new remote copy. When a thread enters a Fetch-Me closure, the closure is converted into a Fetch-Me blocking queue, into which the thread is enqueued and a FETCH message is sent to the global address in the Fetch-Me. On receipt of a FETCH message, the target PE packages up the appropriate closure, together with some nearby graph, and sends this in

a RESUME message to the originator. The originating PE unpacks the graph, redirects the Fetch-Me to point to the root of the graph, and restarts any threads that were blocked on global closures transmitted in the packet.

The virtual shared memory requires global addresses which are (PE, local identifier) pairs. Local identifiers are mapped to local heap addresses using a Global Indirection Table (GIT). This ensures that the local identifiers are immutable and enables the use of a local mark and copy garbage collector which takes the entries in the GIT as additional roots. A weighted reference counting algorithm is used to perform global garbage collection.

In order to transfer a subgraph from one PE to another, GUM uses sophisticated packing/unpacking algorithms, which guarantee that all the links back to the original graph are maintained and that the duplication of work is avoided. Packing proceeds closure by closure, breadth-first into a single packet. It stops, when all reachable graph has been packed or when the packet is full. When a packet is full, outstanding closures get global addresses and are packed as Fetch-Mes. Unevaluated closures and black holes (`queueMes`) also have to be packed as Fetch-Mes.

GUM uses a passive work distribution scheme. PEs with an empty pool of running threads and an empty pool of initiated "parallel" expressions start to ask at random another PE for work by sending a FISH message.

8.6.2 Reusing and simplifying the GUM system

In principle, the GUM system is more complex than what is needed for Eden, as it supports a virtual shared memory in which the global program graph resides and is completely demand driven. But the overall organization of the system, the interleaved evaluation of independent threads on a single processor and the graph packing and unpacking algorithms can be incorporated in the Eden system, although they are used in different contexts. Each PE is modelled by an MPI process and executes a set of Eden processes.

Thread management.

GUM's thread management is perfectly appropriate for Eden's threads. The sequential evaluation of threads and the synchronisation of threads within the same process need not be changed. The TSOs are extended by the outport specification which consists of a local outport identifier and the global address of the corresponding inport to which the result of the thread's computation must be sent. For simplicity the threads of all processes which are executed on the same PE share the heap and the scheduler.

Memory management.

GUM's global addresses are used for references from outports to inports. A global address consists of the identifier of a PE, i.e. the rank of the corresponding MPI process, and an identifier which is unique within this PE and will be used to index the inport table (see below) which is similar to the GIT.

There is no need for global garbage collection in the Eden runtime system. The local garbage collection proceeds as in the sequential runtime system. The only extension necessary is the use of the outport addresses (contained in a runtime table, which is explained below) as additional roots in the marking phase. This extension is similar to

the organization of local garbage collection in the GUM system where the entries of the global indirection table (GIT) are used as additional roots for the local garbage collection.

Graph Packing/Unpacking.

Eden's runtime system uses three primitive functions *sendVal*, *sendHead* and *closeStrm* implemented by new routines for triggering communication. The routines have to transfer a part of the heap (the result of the evaluation) to the global address given in the outport specification of the executing thread. In order to pack the subheap into a packet that can be sent across the network, GUM's graph packing and unpacking algorithms can be used. In Eden it is ensured that a subgraph to be output contains only normal form data. This allows a substantial simplification of GUM's packing algorithm which checks for each closure node whether it contains normal form data or not in order to avoid the duplication of work. The effort for the globalization of shared non-evaluated closures (thunks) is saved. The GUM algorithms however have to be extended by the possibility to send more than one packet. The routines for *sendVal* and *closeStrm* terminate the executing thread after the transmission.

8.6.3 Extending GUM

In order to establish the communication structure of the Eden process systems, new runtime structures have to be introduced into the GUM system. The following (logically distinct) tables are used in PEs to represent the communication channels and process interface information:

- The *inport table* maps locally unique identifiers of inports to heap addresses. These heap addresses point to `queueMe`-closures which represent the not yet available input. A thread which tries to read such a closure will be suspended until the input data arrives and replaces the `queueMe`-closure. In order to propagate termination of threads, the `queueMe`-closures contain the global address of the outport which feeds them.

- The *outport table* maps locally unique identifiers of outports to a pair of identifiers of the corresponding thread and process. The thread identifier is the heap address of its TSO. The process identifier is an index to the process table. The outport table is used for system management, i.e. garbage collection, termination, error detection etc.

- The *process table* contains for each process identifier the lists of inport and outport identifiers.

The runtime tables maintained by the Eden system are crucial for handling *process termination* efficiently. The TSO addresses within the outport table are additional roots for the local garbage collection. An outport is closed if a thread terminates or if the corresponding inport is closed. Whenever an outport is closed, it is removed from the list of outports in the corresponding process table entry. An inport is closed if the process terminates or if it is not referenced any more. The latter is detected by the local garbage collection which checks whether the `queueMe`-closures of the inports have been reached from the roots. A process terminates when all its outports have been closed. When an

inport is closed, a `terminate` message is sent to the associated outport, the address of which is noted in its `queueMe`-closure. On reception of such a `terminate` message, a PE closes the addressed outport and kills the corresponding thread. This may lead to the termination of further processes.

The closing of inports whose `queueMe`-closures cannot be accessed any more is an important vehicle for the termination of subsystems of processes with mutual input/output connections which are no longer needed. Most of such systems will shut down by the closing of inports, when the outports which connect these systems with the main system will be closed.

Process creation.

When a process instantiation is evaluated, a process must be generated. In this situation, the input expressions and the process abstraction will be available as heap closures, the addresses of which are given on the runtime stack within the thread state object. The local result of the process creation is the tuple of new inports, which will be filled by the child process. In detail, the parent will perform the following steps:

1. Generate `queueMe`-closures for the new *inports*, to which the outports of the new process will point, and provide new identifiers for them and insert them into the inport and process tables.

2. Provide new identifiers for the new *outports* of the current process, which have to be connected to the inports of the new process, and insert them into the outport and process tables correspondingly.

3. Compose a `process` *message*, which consists of the processor identifier, the whole subheap of the process abstraction[4] and the identifiers for the in- and outports of the process. The global addresses of the in- and outports are pairs with the processor identifier and the respective port identifiers. The subheap can be packed using a simplified version of the GUM packing algorithms that need not check for unevaluated parts of the heap.

   Send this message to the PE determined by the process distribution and load balancing algorithm (see below).

4. Generate *new threads* for the evaluation of the input expressions for the new process, and block them until the new process has returned the global addresses of the corresponding inports.

5. Return the tuple with the *heap addresses* of the `queueMe`-closures which represents the (not yet available) result of the subprocess evaluation.

Correspondingly, the PE that accepts such a `process` message will:

[4]If the subheap depends on input from other processes, i.e. it contains a `queueMe`-closure which represents an inport, a runtime error will occur. We plan to incorporate a static analysis in the compiler which checks for global parameters in process abstractions depending on inputs of the generator process in order to avoid this runtime error.

1. Generate `queueMe`-closures for the *inports* of the new process and adapt the inport and process table. The latter is extended by an entry for the new process. The global addresses of the corresponding outports can immediately be written into the `queueMe`-closures.

2. Generate new identifiers for the *outports* and extend the outport and process table accordingly.

3. Unpack the *process abstraction* and start the initial thread of the process evaluation which will build up its common environment and then spawn the threads for the evaluation of the outport expressions.

4. Send an `acknowledgement` *message* to the generator process which contains a mapping between the global addresses of connected in- and outports.

On receipt of the `acknowledgement` message, the generator PE writes the global addresses of the in- resp. outports, into the thread state objects of the outports and the `queueMe`-closures of the inports, thereby reactivating the blocked threads.

Process distribution and load balancing.

Eden requires an active process distribution algorithm. Instantiated processes will definitely be evaluated as a parallel unit. In the first version of the parallel runtime system we use a random distribution algorithm without any bookkeeping about processor loads. A processor that instantiates a new process sends the `process` message to a randomly selected PE. This PE may refuse to accept a `process` message when its workload is high and there is not enough space in the runtime tables to allocate another process. In such a case this processor sends the `process` message further on. We plan to investigate more elaborate versions of distribution and balancing algorithms.

Up to now input/output can only be performed by the main program which has the type `IO ()` as in Haskell. The main program may create processes by process instantiations. These processes evaluate output expressions to normal form and return their values via their outports. They are only able to communicate with each other, but not with the standard environment.

8.6.4 Refinements

Adaptive communication:

For efficiency reasons, it is advisable to adapt the amount of output produced to the demand for information. In particular, we want to avoid that senders flood the heap of slower receivers and that data is sent to receivers which do not need further data any more. It is planned to incorporate a simple ticket-based control mechanism as it has been proposed in [Aßm96].

Bypassing channels:

In many cases communication paths can be optimized by bypassing immediate inport-outport connections within processes. Whenever a process simply copies data from an

inport to an outport without processing the data itself, the connection can be shortcut and the real producer and consumer can be connected directly. This optimization will be based on the compile-time checks sketched in Chapter 6.1.2.

8.7 Related implementations

In this section we will investigate different implementations of concurrent and parallel languages and discuss in how far techniques used there can be used for a parallel implementation of Eden.

Abstract machines

In parallel functional programming, three main directions of research have been pursued [Ham94]: the exploitation of implicit parallelism, the incorporation of parallel operations and the integration of functional and concurrent programming.

With *implicit parallelism* annotated expressions are handled by defining "sparks" (i.e. putting their heap addresses into a work pool), or by adding new threads for their evaluation to the task list. Sparked expressions and threads can be evaluated either locally or remotely, depending on the load balancing algorithm and the current work load. Accordingly, most of the parallel abstract machines for functional languages are based on a low level "fork and wait" mechanism: expressions are spawned for parallel evaluation. Threads which require the results of spawned expressions have to wait for the results to be sent back, see for example [THM+96, Kes96, Cha95, Hor96]. They support a global address space and provide a virtual shared memory. In some implementations, e.g. [THM+96], a request for the remote data must be sent, because results of parallel expressions are not automatically returned to their original location, as e.g. in [Kes96]. While waiting for the data, the evaluation continues with another active thread or a new spark. If the whole evaluation is demand-driven, it is sufficient to use a non-preemptive scheduler.

DREAM differs substantially from these abstract machines, because there is no need for a virtual shared memory. Processes have no direct access to data of other processes. They are closed entities and global data must be explicitly communicated to them. In particular this implies that there is no need for a global memory management. Of course the process interface contains references to other processes, but these are explicit and set up a well-defined communication structure. As communication channels are one-to-one, it is known which data is received by which processor and consequently data can be directly sent there without waiting for a request. We expect this direct way of data exchange to reduce communication costs considerably. Before expressions are transmitted, they are evaluated to normal form. This simplifies the low level data transmission and supports granularity control as it is determined where expressions are evaluated. A fair scheduler is however mandatory for the thread management in Dreams.

Special parallel constructs are provided in data-parallel languages like NESL [Ble96a], SISAL [Ske91] and pH [NAH+95] or skeleton-based coordination languages like SCL [DGTY95]. These languages encapsulate parallelism by providing distributed data structures and predefined parallel operations on these structures. A general process-oriented view of parallelism is not supported. NESL for instance supplies parallel sequences (arrays) and data-parallel operations like apply-to-each (map). The structured coordination language SCL is based on skeletons, i.e. higher order functional forms with built-in parallel

behaviour. These are naturally data-parallel and express data partitioning, placement and movement as well as control flow on a high level of abstraction. As in these languages parallelism is restricted to predefined operations and skeletons, they can be implemented by adding parallel primitives to a sequential implementation. The development of a specific parallel abstract machine is not required. NESL is compiled to VCode, a small stack-based intermediate language with highly optimized functions which operate on sequences of data [BCH+94] and SCL is compiled to Fortran plus MPI.

A coordination language which is closer to Eden is *Caliban*[Kel89]. In correspondence with Eden, one can define networks of processes which communicate via head-strict lazy streams. The main idea in both approaches is to make the communication structure of a process system explicit in order to map it directly on a distributed memory multiprocessor. Unlike Eden, Caliban is restricted to static process systems which must be configured at compile-time. Only basic values can be exchanged between processes and each process has only one output stream. There are no special constructs for reactive systems.

Caliban provides annotations for partitioning a program into a process net. They can be defined by functions, which are partially evaluated during compile time until the annotations are in primitive form. The Caliban compiler is based on the sequential Haskell compiler of the FAST project[GHW90]. Similarly to our approach, the specific Caliban extensions have been integrated into the base language (Haskell) in order to use as much of the machinery of the base language as possible. Due to the imposed restrictions the Caliban compiler can extract the whole process net information at compile time and translate it into a call to a special system primitive called `procnet` which implements the run-time parallelism. The resulting standard functional program is then compiled like any other for each processor element. The implementation of the `procnet` function which is called first on each processor element sets up the process network to start computing and afterwards controls interprocessor communication. Each process is evaluated eagerly by its processor element. Its evaluation can be blocked by an inter-processor data dependency or when the output buffer space is used up. Like in Eden, buffering is provided naturally by the heap.

Because Eden supports dynamic process creation and several outputs of processes, it is not possible to implement process creation and communication by a single primitive function with an interface to a purely sequential runtime system.

Most *concurrent* functional languages like Facile [GMP89], Concurrent ML [Rep91], Erlang [AWV96] and Concurrent Haskell [PGF96] provide low level parallel extensions to functional languages, which are implemented on a shared memory base using fair schedulers to switch between the concurrent activities. Distributed implementations are feasible, but require the implementation of a virtual shared memory as in the runtime systems of functional languages with implicit parallelism. The low level process model of these languages obstructs the abstraction of a communication structure which could easily be mapped to a distributed system.

Runtime systems

Parallel functional runtime systems like Haskell's GUM (Graph reduction for a Unified Machine model) [THM+96] and CLEAN's PABC (Parallel ABC machine) [Kes96] typically allocate one logical processor element per physical processor. The logical processor elements act as evaluators for parallel expressions. They maintain a pool with tasks which

may be migrated to other processor elements. Each task corresponds to the evaluation of an expression to weak head normal form. Task distribution is usually done on demand, i.e. processor elements with empty work pools send requests for work to other processor elements. The runtime structures of tasks under evaluation are heap-allocated. Each logical processor element is capable of executing several tasks at the same time. A switch to another task occurs whenever the task under evaluation is blocked e.g. because it needs the value of some non-local subexpression.

The systems provide a global address space, sometimes called virtual shared memory. Whenever the evaluation needs the contents of a global address, a request message is sent to the processor element holding that address. In the PABC machine such request messages trigger the evaluation of the expression at the global address. In GUM the graph at the global address is packed and transferred to the requesting processor element.

The exploitation of implicit parallelism is a neat and simple idea, the implementation of which however requires powerful and complex runtime systems. Unfortunately it turns out that the correct and effective placement of annotations by programmers is not as easy as it seems at first glance, but requires some knowledge about the internals of the parallel runtime system. In [THLP96] undesirable effects caused by the lazy evaluation strategy in combination with parallelism annotations are discussed. Even worse is that in principle the programmer 'does not get what s/he sees'. Reconsider the process ring of Example 1. Although correctly placed annotations allow to spawn a parallel process for each node in the process graph, the communication structure will be completely different. This is due to the fact that the communication channels correspond to parameters, in implementation terms global addresses. I.e. the request/answer mechanism, a two message protocol, will be adopted for 'communicating values along the communication lines' of the process graph. It depends on the runtime system which process will evaluate the expressions to be communicated. Thus it is difficult for programmers to understand what is going on in the parallel system. Parallel profiling tools [Ham97] only help to some extent.

8.8 Conclusions

We have developed an intermediate language PEARL and a parallel abstract machine DREAM. By giving a complete specification of the DREAM extensions to the STG machine, we provide a concise abstract view of Eden's parallel runtime system.

In comparison with other parallel abstract machine models for functional languages, substantial simplifications in the parallel runtime system are possible due to Eden's explicit way of expressing parallelism and process interaction. Since one of Eden's design goals is to be efficiently implementable on distributed memory machines, there is no global program graph and thus DREAM dispenses with virtual shared memory and global memory management.

In Eden, the receiver of some output is always known because channels in Eden are unidirectional and one-to-one. Consequently, communication costs can be kept lower than for demand-driven frameworks, by transferring data to the respective receiver without explicit requests. In addition, outputs are only transmitted in normal form.

In being an abstract machine, DREAM does not address practical details, e.g. the details of message passing. In DREAM, buffer capacity is assumed to be unbounded, but in the implementation special system messages will throttle the sender if the receivers'

buffer capacity is approaching its limit. We have shown that these issues can be handled in the desired way, by describing a runtime system based on this abstract machine. In this way it is demonstrated that Eden's evaluation model forms a comfortable compromise between the efficiency of speculative parallelism and the flexibility and safety of lazy evaluation.

As Eden extends Haskell by a coordination language which allows the explicit definition of parallel process systems, an obvious approach to the implementation of Eden is to extend a Haskell compiler and runtime system. We have shown that the front end of the Glasgow Haskell compiler can be reused for the compilation of Eden into Core Haskell. The back end of the compiler and the runtime system require changes, which are however designed as orthogonal additions to the existing implementation.

Part III

Applications of Eden

Chapter 9

Transformational Programming in the Large

In this chapter, we will discuss the application of Eden to transformational problems. We will present a general methodology for rapid prototyping and software development and exemplify this by applying it to problems from numerical linear algebra.

9.1 The representation of arrays

The most natural representation for matrices and vectors, at least for dense structures, are of course arrays. Due to the combination of single-assignment property and automatic memory management, the integration of arrays is less natural in functional languages than in imperative ones.

Obviously, there are two alternative ways how to handle this conflict between (lazy) functional languages and array computations: On the one hand, one can provide special implementation support for arrays at the expense of language flexibility or expressibility.

- A number of languages support very powerful operations on arrays, but restrict the features of the language considerably[Ske91, Sch96b].

- Another form of special implementation support for arrays is to provide update in place mechanisms. The Glasgow Haskell system offers a variety of array implementations which must be explicitly selected by the user, see [GTDoCS98, Section 4.3].

  In Clean, uniqueness typing is used in order to find out whether there are multiple references to an object[1] The papers [vG97, Ser97b] describe efficient array computations.

On the other hand, one can decide against a reduction of expressibility or classify update in place as "overly imperative" in style and strengthen the research for alternative algorithms that do not rely on it. The algorithms most frequently used in practice today are iterative. This is a drawback with respect to both parallelization on (large-scale) distributed memory

[1] Note that the analyses required for runtime bypassing is related to the ones used in uniqueness typing: one has to find out whether there are multiple references to a particular structure. If there are not, the original structure does not have to be kept any longer.

systems and implementation in functional languages. It is likely that completely different algorithms exist, which alleviate both parallel and functional programming of numerical applications. This direction is pursued by the following approaches:

- In [OW98] arrays of so-called *fat elements* are proposed. They comprise multiple versions of an array, distinguished by unique version stamps. Experiments show the time complexity to be acceptable.

  A similar approach is taken in [EL98].

It is our main goal to retain the full flexibility of lazy functional languages for prototyping purposes. There are a number of powerful mechanisms for stepwise optimization of Eden programs, which will be discussed in Chapter 9.5.2. [Ser97b, vG97, Sch97a] are restricted to the sequential case. It has to be investigated in how far the results obtained there carry over to a distributed setting. For the use of arrays in a distributed setting, the following additional aspects have to be taken into account:

- how well can such an array be *transmitted* between Eden processes?
- how well can such a structure be *decomposed* in order to perform operations on a matrix in a distributed way?

In the following, we will discuss alternative representations for special cases of arrays. The aspect of data decomposition will be discussed in Chapter 9.3.

9.2 The representation of sparse matrices

In this section, we will discuss the different requirements for efficient data structures in functional and imperative settings. In general, the following possibilities for the storage of sparse matrices are considered (see e.g. [Saa96]):

- coordinate format
- compressed sparse row or column format
- diagonal format

These schemes were developed for imperative languages with static arrays and destructive updates. But in a functional setting, it is much more promising to use completely different schemes, instead of optimizing for the above ones. We will discuss the following alternative representations specifically suitable for declarative languages (proposed in [WS92, GSWZ96, Wis92] and others):

- quadtree representation
- binary tree representation
- run-length representation

These representations will be discussed in the following. Alternatively, one could dispense with an explicit storage of the matrix completely and use

- an algorithmic or functional representation

9.2.1 Coordinate format

The coordinate format represents a matrix as a combination of three vectors, one for the nonzero entries, and two for the corresponding row and column indices. The function `coord` below transforms a matrix into such a triplet of vectors. It preserves the ordering of the association list given as the first argument and therefore can be used to implement both row-sorted and column-sorted coordinate format. The function `listArray` is a predefined array function and it transforms an index specification and a list of entries into an array. The auxiliary function `coordL` traverses the association list and inserts the elements of the entries into the association lists for the corresponding vector.

```
coord :: Num a => [((Int,Int),a)] -> Int ->
                  (Array Int a, Array Int Int, Array Int Int)
coord assocList numNZ =
   let (vList, iList, jList) = coordL assocList
   in (listArray (1,numNZ) vList,
       listArray (1,numNZ) iList,
       listArray (1,numNZ) jList)
coordL []                  = ( [], [], [])
coordL (((i1,j1),v1):rest) = (v1:vRest, i1:iRest, j1:jRest)
       where (vRest, iRest, jRest) = coordL rest
```

9.2.2 Compressed sparse row format

The compressed sparse row format again uses three vectors for the storage of the matrix, but compresses the vector representing the rows by storing only the indices which mark the beginning of a new row. Therefore, three vectors `v[1..numNZ]`, `j[1..numNZ]` and `i[1..(n+1)]` are used.

The following function `csr` has as its first argument an association list of a *sparse* matrix (i.e. without zero entries) which is sorted by rows. It calls `csrL` which yields a triplet of lists `(vList, jList, iList)` which it converts into arrays by the predefined function `listArray`. The `csr` format uses the convention that both the first and last boundary is stored in the row field, which is easily achieved by prepending the index `1` to `iList`. The counter `jIndex` is used to store the next free index position in the arrays (here: lists) for `v` and `j`.

```
csr :: Num a => [((Int,Int),a)] -> Int -> Int ->
                (Array Int a, Array Int Int, Array Int Int)
csr assocList numNZ numI =
   let (vList, jList, iList) = csrL assocList 1 1
        in (listArray (1,numNZ) vList,
            listArray (1,numNZ) jList,
            listArray (1,numI)  ([1] ++ iList))
csrL []                   _    jIndex = ( [], [], [jIndex])
csrL (((i1,j1),v1):rest) iOld jIndex =
     if i1 == iOld
     then -- still in the same row of the matrix
       let (vRest, jRest, iRest) = csrL rest iOld (jIndex + 1)
       in (v1:vRest, j1:jRest, iRest)
     else -- new row : iOld has to be incremented
```

```
    let (vRest, jRest, iRest) = csrL rest i1   (jIndex + 1)
    in (v1:vRest, j1:jRest, jIndex:iRest)
```

The above representation is of course possible in Haskell, but it is not a very natural choice. In a functional language, it is much more natural to use a *vector of lists* instead. Each row of the matrix is represented by a list that contains pairs (v_k, j_k) of matrix entries and corresponding column numbers. If `c` is the result, `c!!rowIndex` is the list of entries in row `rowIndex`.

```
csrL2 :: Num a => Int -> [ ((Int,Int),a) ] -> [[ (Int , a) ]]
csrL2 n list = [(map elimRow (filter (sameRow row) list)) | row <- [1..n]]

sameRow i ((i1,j1),v1) = i1 == i
elimRow   ((i1,j1),v1) = (j1,v1)
```

Matrix - vector multiplication using csr. The function `prodCsrL` computes an inner product of a list of tuples (supposed to be a line of a matrix as generated by `csrL2`) and a vector that is represented as a Haskell `Array`. For technical reasons, the list is used as second argument and the vector as first argument. This kind of matrix - vector multiplication is called *inner product form* (cf. [Saa96]). The function `prodCsrL` can easily be used with higher order functions both for the representation as list of lists and as vector of lists.

```
prodCsrL _ [] = 0
prodCsrL vector ((j,v):list) = v * vector!j + prodCsrL vector list
```

The compressed sparse column format can be implemented analogously.

The diagonal format can be implemented in a way similar to the ones above. However, it is obvious that the above storage formats are not well suited to a functional setting, with regard to both programming convenience and efficiency.

9.2.3 Quadtree representation

In the quadtree representation, a square matrix can on the top level consist of a `Diag` node or a `Quad` node. `Diag` is used for diagonal matrices with identical values. It is a unary data constructor which takes as its argument this value. The information that the non-diagonal values are zero is implicit. All matrices which do not have this special form, are represented by `Quad`. This is a data constructor that takes four arguments, namely the representations of the four quadrants. The size of the matrix is implicit in this representation, it is only specified on the level of the enclosing definition of `Qmatrix`.

```
data Quadtree a = Num a =>
                  Quad (Quadtree a) (Quadtree a) (Quadtree a) (Quadtree a)
                  | Diag a                                deriving (Eq, Show)
data QMatrix a  = Num a => Mat Int (Quadtree a)           deriving (Eq, Show)
```

In declarative settings, this storage scheme is the most successful one. Especially for parallel computations, it is very useful, because it can be decomposed efficiently. The decomposition of quadtrees will be discussed further in Chapter 9.3.

Alternatively, a *binary tree* with entries only on the leaves can be used to represent a row of a matrix. The matrix itself is then a list of such rows (plus its dimension). Analogously to the quadtree, the binary tree cuts the present structure in half and represents the subtrees in a scalable way. This means, this representation is a quadtree representation restricted to only one dimension. In [WS92] it is shown to be superior to the quadtree representation for the SOR algorithm. This is due to the fact that SOR requires frequent access to isolated rows of a matrix, which is expensive with quadtrees. For other algorithms, however, quadtrees outperformed binary trees.

9.2.4 Run-length representation

This representation parts with the symmetric dissection of matrices into substructures and compresses arbitrary neighboring identical values. The underlying idea is to compress a sequence of identical values by specifying a so-called run-pair with the repetition factor and the values as its components. In this way, sparse vectors can be represented by packing subsequent zeroes into one run-pair and by using separate run-pairs for the (different) nonzero entries. As this representation is more costly than the 'naive' list representation for dense vectors, the encoding by a list of entries is added as an alternative (which is used for parts, not for the whole vector). In this representation for vectors, the row indices are implicit (i.e. the elements have to be counted), only the overall number of elements in the vector is given explicitly and does not have to be counted. Note that for a matrix, a list structure is used for the rows.

```
type VecRunLen = ( Int,  [RunLen]  )
type MatRunLen = ( Int, [[RunLen]] )
data RunLen    = Num a => RunPair Int a  |  List [a]    deriving (Eq, Show)
```

9.2.5 Algorithmic resp. functional representation

If a matrix is sparse and forms a regular pattern, the representation by a *mapping* is a reasonable choice. Such representations are in use in imperative settings as well[Sch97b].

Note that one is not restricted to completely static mappings, because λ-abstractions can be generated at runtime. But matrices represented in this way could not be overwritten, because update in place for a λ-abstraction (or function) in Haskell is not possible in the current implementation. But for algorithms with "constant" sparse matrices with repeating patterns, such a representation could be very efficient.

Its main advantage is that for suitable matrices it consumes less space than a *storage* scheme. As arrays in Haskell are interpreted as a special kind of mapping anyway, this representation is not considered further.

Summary. The above discussion has given a short overview over possible ways for the encoding of sparse matrices. The storage format most useful for a certain application greatly depends on the characteristics of the matrices and the algorithm. For special classes of matrices, modifications or combinations of the above data structures could give better performance. For these reasons, it is vital to be able to experiment conveniently with different representations. This aspect will be addressed in the next section.

9.3 Distributed data structures

In the previous section, we have presented a number of different representations for matrices, ranging from different predefined Haskell arrays to special user-defined data structures. In order to handle them in a convenient way in a distributed setting, we introduce a type class `Dist` for distributed matrices and vectors which provides flexibility in the data structures used. The overloaded operations `compose` and `decompose` are defined for different structures and parameterized for different granularities (see below):

```
class (Num a) => Dist a where
  compose ::   Int -> [a] ->  a
  decompose :: Int ->  a  -> [a]
```

In Eden, processes are defined explicitly, along with the data to be consumed by them. In this way, *data distribution* can be defined. The programmer can then work with a generic decomposition – computation – (re-)composition scheme by selecting (or defining) one from a number of frequently used decomposition schemes for this data structure. For example, for a quadtree one can use `(de)compose 2` (resp. `(de)compose 4` etc.) for the straightforward decomposition into 4 (resp. 16 etc.) quadrants:

```
instance (Num a) => Dist (Quadtree a)
  where
  compose 2 (x11:(x12:(x21:(x22:[]))))
      = if and [(isDiag x11), (x11 == x22),
                (x12 == (Diag 0)), (x21 == (Diag 0))]
           then x11
           else Quad x11 x12 x21 x22

  decompose 2 (Quad x11 x12 x21 x22) = [x11,x12,x21,x22]
  decompose 2 (Diag k) = [(Diag k), (Diag 0), (Diag 0), (Diag k)]

isDiag (Diag _)       = True
isDiag (Quad _ _ _ _) = False
```

In order to be able to work with matrix-vector operations on the substructures, the same decomposition parameter is used for the matrices and the vectors. Alternatively, we could define versions which cut the matrix in halves vertically (which is better than horizontal for the matrix vector multiplication), `(de)compose 3` which decomposes a sparse matrix into three blocks: x11, x22, x12+x21. In Chapter 9.4.2, we will present an example, where the most appropriate scheme is selected automatically, by computing the first argument of `(de)compose` by a function `granularity`.

Other high-level models for distributed structures can be found in [BK95]. Sophisticated distribution schemes for specific algorithms are for example developed in [Bas97]. An interesting special case form replicated schemes such as described in [SPOP97, PS97, Ser97a] which of course can be expressed in Eden as well.

9.4 Solving linear equations

In this section, we will present a case study on linear equations (see e.g. [Sto89, SB90]).

9.4.1 Direct methods for the solution of linear equations

LU decomposition.

$$r_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} r_{kj} \text{ for } j = i, \ldots, n$$

$$l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} r_{kj}) / r_{jj} \text{ for } i = j+1, \ldots, n$$

```
lr :: Array (Int, Int) Double -> Array (Int, Int) Double
lr a = (array (bounds a)
        ([ ((i,j), do_r i j) | i <- [1..n], j <- [i..n]  ] ++
         [ ((i,j), do_l i j) | i <- [1..n], j <- [1..i-1]  ] ))
       where
       do_r i j = a ! (i,j) - sumM (1, i-1)
                            (\k -> (lr a) ! (i,k) * (lr a) ! (k,j))
       do_l i j = ( a ! (i,j)
                   - sumM (1,j-1) (\k -> (lr a) ! (i,k)* (lr a) ! (k,j))
                  ) / (lr a) ! (j,j)
       sumM (a,b) f | a > b = 0
       sumM (a,b) f         = f a + sumM (a+1, b) f
```

Loidl et al. found that LU decomposition does not offer sufficient potential for parallelism. Because of this negative result, other methods for the *exact* solution of linear equations are considered in[LHP94, Loi97b].

Even though LU decomposition is not directly parallelizable, it may be interesting as a basis for more complex methods, such as parallel algorithms based on multi-frontal elimination[DR83]. It is considered here because it exposes the benefits of lazy evaluation very clearly. In [GSWZ95], lazy evaluation in the Cholesky algorithm, which has similar characteristics as LU with respect to parallelism, has been found to be essential to parallel performance in a finite-element program.

9.4.2 Iterative methods for the solution of linear equations

Conjugate Gradient algorithm. The conjugate gradient (CG) algorithm is an iterative method for solving large linear systems. It proceeds by generating vector sequences of iterates (successive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. The following is a Haskell version of the CG algorithm (cf. the Miranda presented in [WS92]). A data type `ISol` is used for representing an iterative solution, consisting of an iterate `x`, a residual vector `r`, the search direction `p` and an iteration counter.

```
data ISol = IterSol Vector Vector Vector Int       deriving (Eq, Show)

cg :: MatF -> ISol -> ISol
cg    a       init = until converge (nextIter a) init

converge :: ISol -> Bool
```

```
converge (IterSol x r p k) = dotProd r r < epsilon

nextIter :: MatF -> ISol -> ISol
nextIter a ( IterSol x  r  p   k)
         = ( IterSol x' r' p' (k+1))
         where
         q     = matvec a p
         pq    = dotProd p q
         rr    = dotProd r r
         qq    = dotProd q q
         alpha = rr / pq
         beta  = alpha * qq / pq - 1
         r'    = r  - scalMult alpha q
         x'    = x  + scalMult alpha p
         p'    = r' + scalMult beta p
```

[Ser97b] describes the programming of the conjugate gradient algorithm in **Clean**. A first straightforward implementation is very easy to read, but is approximately twice as slow as a C implementation. The speed of C can be approximated rather closely, but only with optimizations which make the program less readable: map fusion and update in place. The changes required due to uniqueness typing clutter up the program. The above Haskell program can of course be optimized as well, but the focus of our case study lies with *parallel computation*. Therefore we develop a parallel version in the following. For possible optimizations of both the sequential and the parallel version, we refer to Chapter 9.5.2.

For the parallelization of an iteration algorithm, it is particularly important that the processes involved are only created once for the complete iteration and not for every iteration anew. Eden differs from implicitly parallel functional languages in that it makes the specification of such stable process systems possible. Naturally, such a solution in Eden uses mutual recursion over streams. We obtain a clear and reusable solution by using a skeleton for this kind of iteration algorithm (see also Figure 9.1):

1. skeleton `parIter` for parallel iteration with repeated matrix-vector multiplication:

```
parIter :: Mat -> ISol ->                          -- matrix, init. appr.
           (Int -> Mat -> [<[Vec]>] -> [[Vec]]) ->              --  mvIter
           (Int -> ISol -> Process [<[Vec]>] ([<[Vec]>], ISol)) --iterProc
           -> ISol                                             -- solution
parIter a iniSol@(IterSol _ _ p _) mvIter iterProc
   = finalSol
     where
     mvOut               = mvIter  gran a (zipWith (:) iniVectors iterOut)
     (iterOut, finalSol) = iterProc gran iniSol # mvOut
     iniVectors = rep gran (decompose gran p) -- list of initial vectors
```

   There are various alternatives where the respective granularity can be determined, e.g. the following extension could be made to the above skeleton:

```
gran = granularity a p   -- depends on matrix structure and
```

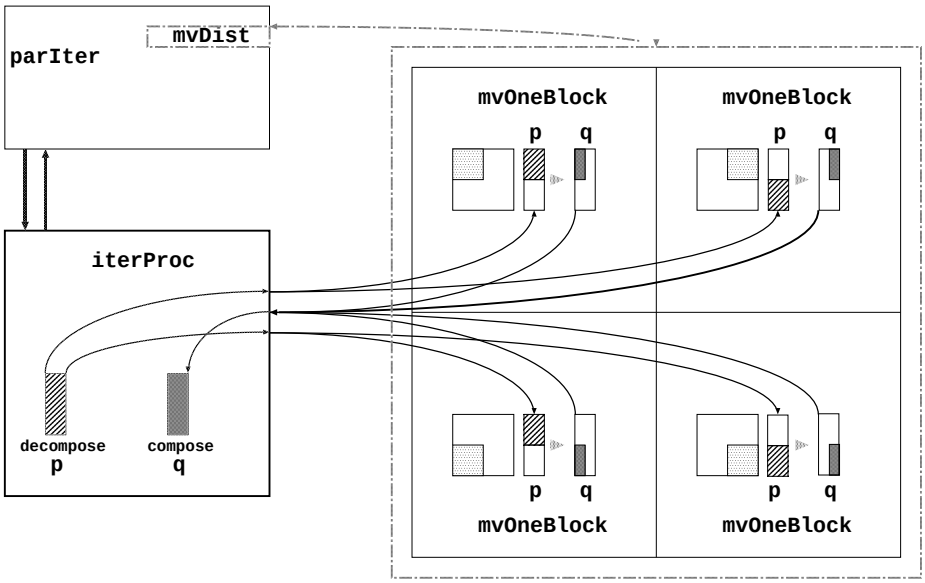


Figure 9.1: Parallel CG algorithm with 2 ×2 processes for matrix vector multiplication

```
                                  -- dimension (derived from vector dimension)
granularity _ _ = ..              -- num parts of vector
```

2. The type class `Dist` for distributed matrices and vectors provides flexibility in the data structures used (see Chapter 9.3).

3. Process abstraction `iterProc` defines a scheme for an iteration process that communicates with `g*g` processes[2] for matrix-vector multiplication, composes the result vector `q`, calls function `nextIterRest` which performs the remainder of the iteration algorithm, and decomposes and distributes the new vector `p'`:

```
iterProc :: (Vec -> ISol -> ISol) -> Int -> ISol ->
            Process [<[Vec]>] ( [<[Vec]>], ISol )

iterProc nextIterRest g oldSol
   = process qInput -> iterFct oldSol qInput
     where
     iterFct oldSol qInput
       | converge oldSol
       = ( [ [] | i <- [0..g*g-1] ], oldSol )
     iterFct (IterSol x  r  p   k) qInput
       =  ( pResult, finalSol )
       where
         -- RECEIVE q
         (qList,qRests) = splitHead qInput
         q = compose g (map vsum (rows g qList))
```

[2]Explicit index selection has to be used in the definition of the output streams in order to prevent blocking.

```
        -- COMPUTE
        (IterSol x'  r'  p' k') = nextIterRest q ( IterSol x  r  p   k)

        -- SEND p'
        pResult = [ (pList !! i) : (pRest !! i) | i <- [0..g*g-1] ]
        pList = rep g (decompose g p')

        -- recursive call of iterFct:
        ( pRest,  finalSol ) = iterFct (IterSol x'  r'  p' k') qRests
```

The following auxiliary functions are used:

```
splitHead :: [[a]] -> ([a],[[a]])
splitHead [] = ([],[])
splitHead ((h1:t1):rest) = (h1:rest1, t1:rest2)
                           where (rest1, rest2) = splitHead rest

rep :: Int -> [a] -> [a]
rep 1 list = list
rep n list | n > 1     = list ++ (rep (n-1) list)
           | otherwise = error "rep"

rows :: Int -> [a] -> [[a]]
rows g [] = []
rows g list = (take g list):(rows g (drop g list))

vsum :: Num a => [a] -> a
vsum [x] = x
vsum (x:rest) = x + vsum rest
```

4. Finally, `cgNextIterRest` is a function that contains the remaining parts of the iteration:

```
cgNextIterRest :: Vec -> ISol -> ISol
cgNextIterRest     q    (IterSol x  r  p   k)
           = ( IterSol x' r' p' (k+1))
         where
         -- q     = matvec a p (parallel)
         pq    = dotProd p q
         rr    = dotProd r r
         qq    = dotProd q q
         alpha = rr / pq
         beta  = alpha * qq / pq - 1
         r'    = r  - scalMult alpha q
         x'    = x  + scalMult alpha p
         p'    = r' + scalMult beta p
```

Consequently, the parallel CG algorithm is called in the following way: `parIter mat initSol mvDist (iterProc cgNextIterRest)`

5. The function `mvIter` decomposes matrix a and spawns processes for distributed matrix - vector product and process (abstraction) `mvOneBlock` performs repeatedly matrix-vector multiplication of its block of matrix a and a stream of input vectors:

```
mvDist :: Int -> Mat -> [<[Vec]>] -> [<[Vec]>]
mvDist    g      a      pStream
   = zipWith (mvOneBlock #) aBlockList pStream
     where
     aBlockList = decompose g a

mvOneBlock :: Process ( Mat, [Vec] ) [Vec]
mvOneBlock    myA    pInput =  map (matvec myA) pInput
```

This approach works with a flexible number of subprocesses for matrix vector multiplication and the flexible scheme for distribution shown in Chapter 9.3. The function `granularity`, that determines the decomposition, is called at the very start of the algorithm. This decomposition parameter is used for the matrix and the vectors. For an iteration algorithm with repeated matrix-vector multiplication (such as CG), the distribution has to fulfill two requirements: Firstly, the matrix has to be decomposed in such a way that the matrix-vector multiplication can speed up. Secondly, the whole system should be well-balanced, i.e. the parts executed in parallel should be equal in runtime. An investigation of distribution heuristics which take into account characteristics of both the matrix and the underlying machine can be found in [Bas97]. In contrast to the solution presented here, these schemes distribute both matrix-vector and vector-vector operations.

Note that this solution is flexible not only in the *distribution* of the data, but also in the *representation* of it. If a particular algorithm is known to work especially well with a particular matrix representation (as for example CG with quadtrees and SOR with the run-length representation[WS92], optimized iteration processes could restrict the matrix to this type. Consequently, the programmer would only have to specify that the matrix belonged to a class of sparse matrices. The resulting type of the process abstraction would then imply the representation that is most useful in this context and it could be generated with the aid of an overloaded transformation function.

If no reliable 'theoretical' information about the structure of the problem can be exploited, one could of course also resort to a systematic series of tests for all available decompositions.

Outlook on other algorithms. There are various references to further numerical algorithms (not only for the solution of linear equations) which have been successfully treated with functional approaches[3].

For instance, *adaptive multigrid* methods for the solution of partial differential equations are difficult to handle in imperative settings, but have been used successfully in

[3]See also the collection of different approaches `http://www.informatik.uni-kiel.de/~cr/FP.html` by Claus Reinke.

declarative ones, e.g. in SAC[Sch97a, Sch96b] or Skil[BK95, Bot98]. Finite-element computations are described in [LKC93, GSWZ95]. Further scientific computing problems are discussed in [Car93] and [vG97].

In the field of computer graphics, a new version of the *hierarchical radiosity* algorithm that is especially suitable for a functional description and for parallel execution, is described in [OR97]. A functional specification of the *JPEG* algorithm is presented in [Fok95]. This article also emphasizes the aspect that functional programs can serve as executable specifications, which will be discussed in detail in the following.

9.5 Software engineering and rapid prototyping

9.5.1 Software engineering

Throughout this section, we have emphasized the "prototyping" aspect of Eden. Our main goal is to provide an environment for the development of parallel algorithms, which is as flexible as possible. In the following, we will sum up in detail the aspects that make Eden valuable for rapid prototyping:

Proceeding from a sequential version of an algorithm in Haskell, how easily and efficiently can this be turned into a parallel Eden program?

Functional programs as executable specifications

In software engineering, the following phases are distinguished (cf. e.g. [Hin92]):

1. Analysis of the problem
2. Design and specification, separation into different modules
3. Validation of the specification
4. Implementation of the specification
5. Test of the implementation

We will now apply this scheme to the specific development of a *parallel* program (which could e.g. be the case when previous tests and implementations made clear that no sequential implementation can deliver the performance required) and analyze how Eden can be used in the different stages.

Analysis of the problem. In this special case, the usual problem analysis does not have to be carried out, because the problem is assumed to be well-studied in a sequential setting.

Design and specification. In this step, the hierarchy of different components will be determined. For this to be possible, profiling of existing implementations in order to find the "hot spots"[4] which are critical for performance. If the existing implementation is imperative, conventional profiling and visualization tools can be used.

[4]In the development of the skeleton solution based on `parIter` presented above, this step involves finding a useful separation between `mvmult` and the rest of the iteration.

In general, we assume that a parallel Eden program will be used as an *executable specification* of the parallel implementation. A first step towards this Eden program would of course be the implementation in Haskell.

Not only is such a sequential version easier to develop than the parallel version, it is also "nearly a parallel Haskell program". Suitably decorated with parallelism annotations or evaluation strategies[5], it can help us to *benchmark the specification.* The article [THL+97] describes the idea to simulate with GranSim the "ultimate" parallel machine with unlimited number of processors, unlimited bandwidth and zero latency in order to find out how much potential parallelism is present in a program.

In our case, this potential for parallelism would not refer to the (final) Eden program we are going to develop, but it would provide valuable insights into the characteristics of different possible solutions. Based on the experiments with parallel Haskell and GranSim, we can then develop the parallel Eden program.

In principle, it is possible that the sequential functional program already introduces unwanted sequentiality constraints, because the sequential program already contains some specification of an unsuitable *algorithm*, i.e. of the "how" and not only of the "what" of the computation. In extreme cases, experiments with other sequential programs, either in Haskell or in dataflow languages like pH[NAH+95], have to be taken into consideration.

Validation The validation is greatly alleviated by the fact that our specification is executable and functional. Formal methods can be applied to it.

Implementation Because the specification is already a form of implementation, in this case the implementation phase degenerates to an optimizing phase. We have to test the performance of the functional parallel prototype and decide which optimizations should be applied. We assume here that there are no constraints as to the programming language used for the final implementation. Consequently, the following options exist:

- The implementation is written in Eden, i.e. the specification is refined (see Chapter 9.5.2 below).
- The implementation is written in Eden, but calls to specific libraries are inserted to optimize performance.
- The implementation is re-written in a different, less abstract language.

9.5.2 Stepwise refinement of a prototype

Eden can be viewed from two different angles: It can serve as a (executable) specification language and it can be used as implementation language. Ideally, we pursue a smooth top-down development from executable specification to an efficient implementation. For this to become possible, the programmer has to have precise knowledge as to which ingredients of Eden are advantageous from which of the two points of view.

We will decompose both Eden and Haskell into a number of constituents, which will be analyzed with respect to both quantitative and qualitative considerations (see e.g. [vG97] for similar analyses for Clean):

[5]In order to make this step as easy as possible, it would be helpful to use methods developed in the context of automatic parallelization.

Qualitative: What are the benefits yielded by this specific language concept?

Quantitative: How much runtime or space could be saved when this feature were not used[6]?

Decomposing Haskell

1. **Lazy evaluation** in many cases introduces a runtime overhead, which one cannot afford in some large-scale application programs. Therefore the use of strictness annotations can optimize the final implementation[HFA+96], whereas the benefits of laziness can be exploited in the executable specification for the parallel program.

   Note that both Glasgow parallel Haskell and Eden also introduce strictness to some extent: Evaluation strategies[THLP98] introduce normal form evaluation and Eden evaluates both top-level process instantiations and outputs of processes eagerly.

2. **Type classes** and overloaded operations belong to the most central features for software engineering . The possibility to define different operations with an identical interface supports modular program development. With multi-parameter type classes[PJM97][7], also operations parameterized over multiple arguments, like matrix-vector multiplication, can be overloaded.

   If implemented using dictionaries, a runtime overhead for overloaded operations is incurred. This can be eliminated by inserting the specific operations by the compiler, which can e.g. be done in Haskell by inserting specialize-pragmas, see [PH97, App. E].

3. **Higher order functions** are another feature which is extremely useful for rapid prototyping, but costly. The runtime overhead can be overcome by removing them by compilation, as is for example described for the languages Skil[Bot98] or **FISh**[JS98a, JS98b][8].

4. **Closures** In general, the representation of data objects by closures is essential for lazy evaluation. But for special cases, this handling can be optimized. Using the GHC, considerable savings in execution time could be achieved by the explicit use of unboxed values, which could be used as first class values[PL91].

5. **Automatic memory management** is vital for programming efficiency[9]. Although indispensable in general, automatic memory management can be undesirable for large data structures which are "modified" frequently[10].

   In these cases, update in place can be induced either by explicit request by the programmer (like in Haskell) or by automatic analyses. Such analyses like *escape*

[6] Or not even implemented at all?

[7] Multi-parameter type classes are available in the GHC from version 3.00 onwards

[8] see also `http://www-staff.socs.uts.edu.au/~cbj/FISh/`

[9] Note that this feature is also included in the imperative language Java.

[10] The case study [Fre96] compares 5 scientific problems programmed in SISAL, in the explicitly parallel imperative language SR[AO91], and in C, run on a shared memory machine with 4 processors. It establishes as one of the main performance problems of SISAL the inability to express that a structure should be updated in place.

analysis can find out data structures which can be safely overwritten, thereby implementing "compile-time garbage collection"[Moh97]. In Clean, *uniqueness typing* is used.

Decomposing `Eden`

1. **Dynamic process creation**, as opposed to skeletons which generate all processes simultaneously, is an expensive feature. Detailed measurements with the parallel implementation of Eden have be carried out in order to quantify this effect.

   Because in many algorithms in this application area the structure of the computation stays the same throughout the whole computation, it would be a waste not to keep the process system. This aspect has been discussed in [BL98]. Program versions with completely static process systems are however more difficult to develop than versions with dynamic process creation. Consequently, a stepwise development of the process system is advisable.

2. **Dynamic handling of channel structures** as opposed to simpler versions described in Chapter 6.2.3 and [BKL97b] can introduce unnecessary runtime overhead. The above paper also proposes specific analyses that help to "bundle" the dynamic splits or replace them at compile time and thereby reduce the overhead.

9.5.3 Summary: Eden for programming in the large

In the above decomposition of Haskell and Eden, we have shown that the features largely considered as "expensive" are very important for prototyping large-scale application programs.

For very specific application areas, it may be the case that certain of these features are no longer helpful. In this case, even a restricted implementation of Eden may be useful. Similar conclusions are drawn in [Sch96b] or in [Ang96], where polymorphism is restricted.

In contrast to this, our aim is to get maximal flexibility and acceptable performance with minimal changes, i.e. without defining a new language specifically designed for scientific computing.

We have shown that Eden can serve as a language for executable specifications and that performance can be optimized considerably by employing optimizations "inside the language". This means, by compilation or by small changes in the programs.

Note that apart from this, performance tuning is also possible "outside the language", i.e. calling special libraries. One can either call C libraries over the C interface of Haskell or one can implement functional libraries[11], which make heavy use of the functional optimizations discussed above. Such libraries could help to hide the expense in terms of programming convenience from the user.

"Parallel rapid prototyping" can be done with Eden: i.e. new parallel algorithms can be developed and tested with Eden and later on be "tuned" by using functional optimizations, libraries or translating them into imperative languages. The following features make this approach viable:

[11] This issue is among others discussed in [GSWZ95].

- Eden programs are shorter and easier to develop than an imperative program.
- Eden programs are clearer and closer to the mathematical representation than imperative ones.
- Eden programs are more flexible than a direct imperative implementation, i.e. they alleviate experimenting with different representations and decompositions of data and different algorithms.

Chapter 10

Simulation and Reactive Systems

10.1 Basic approaches to simulation

Historically, two different forms of simulation have been distinguished: *discrete event simulation* and *continuous simulation*. However, nowadays many authors criticize the strict separation between these two approaches to simulation, see e.g. [Fis95, Poh95].

We will start this section with a short overview over existing languages and tools. There are a number of special programming languages for simulation, such as SIMULA[1] or GPSS. They differ from conventional programming languages in that they offer special support for coroutining and for the handling of queues and time constraints. Additionally, there is a large body of commercial simulation packages available, see e.g. [Wen98] for a short overview.

Simulation with functional languages. Pohlmann[Poh95] postulates the use of functional languages for simulation, because the use of a high level of abstraction reduces the complexity of the simulation. Harrison[Har96] proposes a *simulation monad* that provides facilities similar to SIMULA. Both these articles form interesting starting points for an investigation of the problem domain in a sequential setting.

Parallelism is trivially available. The simulation languages mentioned above necessarily include a coroutining mechanism in order to model the concurrent existence of multiple autonomous components. This aspect can of course be supported even better, if *parallelism* can be expressed.

If we simulate the behaviour of concurrent agents, it is straightforward to model them by separate processes. In addition, there is of course the possibility that individual agents internally contain potential parallelism, because its behaviour is determined by computationally expensive functions.

Our approach to simulation. In the following we will present a set of "agents" which can be used as building blocks for simulations, in the same way as for example the components of the commercial simulation package **Simulink**[2] are used. In this approach, we heavily rely on higher order functions.

[1] See e.g. `http://www.idiom.com/free-compilers/TOOL/Simula67-1.html`.
[2] See `http://www.mathworks.com/products/simulink/`.

10.2 A collection of simulation agents

10.2.1 Agents without internal state

`mapAgent` is an abstraction for an agent without "memory", which maps inputs to outputs according to a unary function specified as parameter. `zipWithAgent` and `zipWith3Agent` are analogous abstractions which work with two (resp. three) input streams and a function with two (resp. three) arguments.

```
mapAgent :: ( a -> b ) -> Process [a] [b]
mapAgent f = process input -> map f input

zipWithAgent :: ( a -> b -> c ) -> Process ([a],[b]) [c]
zipWithAgent f = process (inputA,inputB) -> zipWith f inputA inputB

zipWith3Agent :: ( a -> b -> c -> d) -> Process ([a],[b],[c]) [d]
zipWith3Agent f = process (inputA,inputB,inputC) ->
                          zipWith3 f inputA inputB inputC
```

10.2.2 Agents with internal state

In many cases, what one needs is not an agent that simply maps separate input values to separate output values, but an agent that memorizes certain previous items of information in some internal state. The most straightforward approach to the definition would be the use of the standard function `foldl` in a process, analogous to the ones above. Note however that `foldl` only generates a single result. For simulation applications, it would however be more natural to observe a list of intermediate results. This is expressed by `foldlAgent`, which outputs its current state in every step. If the output to be produced is not identical to the state, `foldlAgentP` can be used. This variant of `foldlAgent` handles a function with a pair of results, namely the internal state plus the protocol entry. It is comfortable to assume the predefined type `Maybe c` for the protocol entries, so that not in every step an entry has to be produced (see also Chapter 10.2.3).

This approach can straightforwardly be generalized to functions with more arguments, as is illustrated by `foldl2Agent`. This agent takes a function with three arguments and produces a stream of intermediate results.

```
foldlAgent ::( state -> a -> state) -> state -> Process [a] [state]
foldlAgent f init = process input -> foldlStrm f init input
             where
             foldlStrm :: ( state -> a -> state) -> state -> [a] -> [state]
             foldlStrm f init [] = []
             foldlStrm f init (x:rest) = let init' = (f init x)
                                         in init':(foldlStrm f init' rest)

foldlAgentP ::(state -> a -> (state,Maybe c)) -> state -> Process [a] [Maybe c]
foldlAgentP f init = process input -> snd (foldlStrmP f init input)
              where
              foldlStrmP :: ( state -> a -> (state,Maybe c)) -> state -> [a]
                             -> (state,[Maybe c])
              foldlStrmP f init [] = (init,[])
```

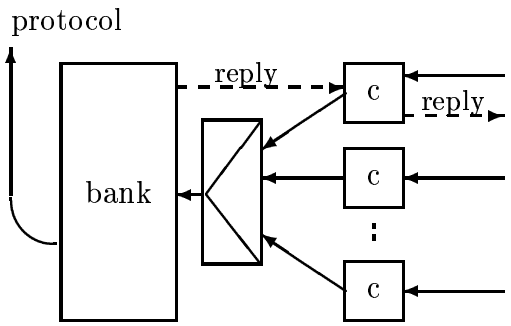
```
              foldlStrmP f init (x:rest)
                                   = let (init',prot) = (f init x)
                                         (iRest, pRest)
                                              = (foldlStrmP f init' rest)
                                     in (iRest, prot:pRest)

foldl2Agent ::(state -> a -> b -> state) -> state -> Process ([a],[b]) [state]
foldl2Agent f init = process (inputA,inputB) ->foldl2Strm f init inputA inputB
              where
              foldl2Strm :: ( state -> a -> b -> state)
                                   -> state -> [a] -> [b] -> [state]
              foldl2Strm f init [] _  = []
              foldl2Strm f init _  [] = []
              foldl2Strm f init (a:restA) (b:restB)
                         = let init' = (f init a b)
                           in  init':(foldl2Strm f init' restA restB)
```

Despite their simplicity, these agents prove very helpful in composing complex simulation programs. This will be shown by the following example (see also [BL98]).

Example 10.1

In the following we will show a simple simulation of a bank process that communicates with a number of customers according to the scheme shown below. Bank customers read streams of messages from the outside world. They react to these messages by sending requests to a bank clerk. These requests contain the id of the bank account, the amount to be deposited and a reply channel. According to the sign of the second argument, such a request is either a query (0), a deposit (positive) or withdrawal (negative) on the bank account specified.




```
data Request = Deposit Int Int (Chan_name BankReply)
```

The bank uses the reply channel in order to send an answer of the following type:

```
data BankReply = BankReply Bool Int
```

where the Boolean tells the customer whether the transaction was carried out successfully and the integer is the new balance of the account after the transaction. The internal state of the bank stores the states of the bank accounts. The bank process outputs a protocol which only lists the accounts that have been modified, together with their new balance:

```
type BankStateArray = Array Int Int
type BankProtocol   = Maybe (Int, Int)

bankProcess :: BankStateArray -> Process [Request] [BankProtocol]
bankProcess = foldlAgentP modifyState
system :: [BankProtocol]
system = bankProcess iniState  # (merge fromCustomers)
          where
          fromCustomers = [ customer i # (toCustomers!!i) | i<-[0..n-1]]
          iniState = listArray (0,n-1) [0|i<-[0..n-1]]
```

Note that this process is independent of the data and their transformation and could easily be defined as a skeleton for a general manager process. A simple function `modifyState` that forbids any overdraft of a bank account, would be:

```
modifyStateO oldSt id value
    | value == 0  = (BankReply True (oldSt!id),  [], oldSt)
    | otherwise   = if newVal < 0
                    then (BankReply False (oldSt!id),  [], oldSt)
                    else (BankReply True newVal, [(id,newSt!id)], newSt)
                    where newVal = oldSt!id + value
                          newSt  = oldSt // [(id, newVal)]
```

The customers in this example are very simple: They have one input channel for the reception of items from the outside world, such as offers to buy goods or messages that they have earned money. The contents of such an item is represented by an integer code for the offer and an integer price. For simplicity, all such items require a Boolean answer, which is also returned using a dynamic reply channel:

```
data Item = Item Int Int (Chan_name Bool)
```

The customers produce a stream of requests sent to the bank. They use some function `accept` in order to decide how to react to incoming items. For accepted items, requests to either withdraw or deposit money are generated.

```
customer :: Int -> Process [Item] [Request]
customer k
   = process fromOutside -> handle fromOutside
     where
     handle [] = []
     handle ((Item x v itemCh):rest)
        = if (accept x v) then                 -- ask bank if possible:
                             new (cName, cReply)
                                     (Deposit k v cName) : wait cReply
          else itemCh !* False par handle rest -- dismiss item
          where wait (BankReply b i) = itemCh !* b par handle rest
```

Note that in applications like the above, the state may become arbitrarily complex and the output of the whole state in every simulation step may not be the desired behaviour. Therefore we prefer the variant `foldlAgentP` of our agent that produces a separate protocol. Correspondingly, `modifyState` produces state and BankProtocol as its result in order to be usable in connection with this agent.

The handling of the communication with the customer is done inside the function `modifyState`. Note that in Chapter 3.3.3, we have found dynamic channels to be safe with respect to nondeterminism.

```
modifyState :: BankStateArray -> Request -> (BankStateArray,BankProtocol)
modifyState oldSt (Deposit id1 v1 ch1)
       | v1 == 0   = ch1 !* (BankReply True (oldSt ! id1))
                      par (oldSt, Nothing)
       | otherwise = if newVal < 0
                      then ch1 !* (BankReply False (oldSt!id1))
                           par (oldSt, Nothing)
                      else ch1 !* (BankReply True newVal)
                           par (newSt, Just (id1,newSt!id1))
                     where newVal = oldSt!id1 + v1
                           newSt  = oldSt // [(id1, newVal)]
```

For each element of a given list `toCustomers` of input streams for customers, the system creates a customer process. It connects them to a bank process which due to the use of `merge` handles the requests in the order of their arrival. The array `iniState` assigns each of these customers one empty bank account:

```
bankProcess :: BankStateArray -> Process [Request] [BankProtocol]
bankProcess = foldlAgentP modifyState
system :: [BankProtocol]
system = bankProcess iniState  # (merge fromCustomers)
          where
          fromCustomers = [ customer i # (toCustomers!!i) | i<-[0..n-1]]
          iniState = listArray (0,n-1) [0|i<-[0..n-1]]
```

◁

10.2.3 Optimistic versus conservative approach

Our simulation example above imposed strict time steps on the whole simulation. For some applications, it is not natural to return a result in every step. In general, there are two approaches how to handle steps of time without any observable change:

- the use of null-messages for points of time without events (the conservative solution in [Poh95]).
- the use of no message for this case (the optimistic solution in [Poh95])

In our definition of `foldlAgentP`, we have used the conservative approach. If one wanted to change to the optimistic one, one would have to replace `prot:pRest` by `optCons prot pRest` in the last line. The predefined "cons" constructor : corresponds to the "conservative cons", of course. The optimistic version is:

```
optCons :: (Maybe a) -> [Maybe a] -> [Maybe a]
optCons  Nothing  rest = rest
optCons  (Just x) rest = (Just x):rest
```

10.2.4 Handling of queues and time dependencies

The handling of queues is a typical task for simulation languages. The merge process abstraction trivially defines a queueing agent. For certain simulation applications, stronger forms of time dependencies may be needed, e.g. an agent that enqueues arriving items in their exact order of arrival. Such an extension of the language could be defined easily.

10.2.5 Integrators

The most important parts in simulators for technical parts are integrators.

Runge-Kutta algorithm. Below we define a function `runge_kutta` which implements the Runge-Kutta method (see [SB90]):

```
runge_kutta f h y x = y + (k1 + 2*k2 + 2*k3 + k4)/6
                      where k1 = h * (f  x         y)
                            k2 = h * (f (x + h/2) (y + k1/2))
                            k3 = h * (f (x + h/2) (y + k2/2))
                            k4 = h * (f (x + h)   (y + k3))
```

The equation $y' = x + y$ with $y(\texttt{x0}) = \texttt{y0}$ can then e.g. be approximated between `x0` and `xn` with step `h` by `foldlAgent (runge_kutta f h) y0 # [x0,h..xn]`, where `f = \x -> \y -> x + y`.

Apart from the above simple process for the treatment of ordinary differential equations, more powerful schemes for the handling of complex systems of differential equations could be defined in Eden, if need be. For example, [LKC93, GSWZ95] describe the functional implementation of finite element methods both in a sequential and in a parallel setting.

10.3 Advanced approaches to simulation

In the previous section we have discussed how the basic components of a simulation environment could be expressed in Eden. For specialized applications, a number of more general approaches to simulation have been developed. In this section, we will present some of them in short and investigate their use in Eden.

Qualitative simulation.[CK94] is an approach that aims at a level of abstraction higher than that described before.

Hybrid discrete event - continuous simulation. A number of approaches aim at the integration of simulation paradigms, e.g. the `SHIFT` software[3] for the modelling of hybrid systems. It supports both continuous and discrete steps and offers dynamic process management on actors. It uses a Runge-Kutta algorithm to integrate the differentially defined variables *simultaneously*.

[3] `http://www.path.berkeley.edu/shift/`

Symbolic dynamics. This techniques reduces a time series to a sequence of symbols. Naturally, such a transformation can be described in Eden in a very intuitive way by an agent with a "memory" and a function. The function maps values to symbols in dependence of the difference between the value and its predecessor.

This function is not injective, but it maps a large number of values to the same symbol and thereby reduces the complexity of the data to be handled, see e.g. [Sud96].

Summary and outlook

In this chapter we have illustrated Eden's approach to simulation with a number of flexible skeletons for simulation agents. Due to polymorphism and overloading, many such functions can be defined independently of the concrete input and output type. This in particular applies to the conservative vs. optimistic view of discrete event simulation: A filter can filter out the zero-messages, if in the consuming component an optimistic view is taken. Lazy evaluation proves to be very useful not only in this case, but also e.g. for numerical simulations which stop upon convergence.

In general, skeletons (higher order functions) for simulation agents provide a very abstract approach to simulation. If for example an additional noise input is to be inserted into a system, a suitably defined combinator (agent) can add this in a way that the rest of the system does not have to be changed.

These agents would be particularly useful if they were accompanied by a library of functions for instantiation and by a graphical interface. An editor in the style of commercial simulation packages would make the overall structure of a simulation clearer for users not familiar with functional programming, while at the same time retaining its full flexibility and the benefits of functional languages.

A short overview over current trends in simulation shows a trend to more flexible and more abstract approaches. We plan to employ such techniques in a medical simulation program for respiratory sinus arrythmia (see [SDA95, Sch94, Sud96]) written in Eden.

Chapter 11

Conclusions

Most parallel programs in use today are written in low-level, imperative languages. In the last decade, it has become clear that progress in parallel programming is blocked by the lack of suitable software. Parallelizing sequential programs by inserting parallel constructs is very complicated. In many cases, sequential programs have to be rewritten completely in order to obtain efficient parallel ones.

On the other hand, there are programming models where the parallelism is (largely) implicit. For real-world applications, the success of these approaches up to now has been limited, because process granularity and communication could not be influenced. In order to alleviate the development of efficient parallel programs, we need programming models which are high-level, but not implicit. The language Eden presented in here is such a model.

11.1 Programming in Eden

Eden is based on the lazy functional language Haskell and introduces the notion of a *process* explicitly. The topology of a process system can also be defined explicitly and both kinds of information are expressed in a functional way. Eden offers a number of valuable features which make Eden a reasonable choice for parallel program development:

- It inherits the benefits of Haskell without any restriction: polymorphic Hindley-Milner type system, type classes, high level of abstraction, and side-effect freedom.
- Rapid prototyping is made especially attractive due to the possibility to apply step-wise refinement using both functional optimizations and a C interface.
- It reconciles laziness and parallelism: the flexibility of lazy evaluation is retained while at the same time a reasonable degree of speculative parallelism is supported.

Additional language features. In addition to the components mentioned above, Eden provides dynamic reply channels which make communication topologies more flexible and allow a comfortable handling of time-dependencies without introducing nondeterminism.

Encapsulation of nondeterminism. In Eden, special nondeterministic processes can be employed in order to obtain more flexible or more efficient programs. This use of nondeterminism does however not impair the possibilities for equational reasoning in the functional part.

11.2 Implementability

The design of Eden has not only advantages from the programming point of view, but also from the implementation point of view:

- Eden processes have a clear interface. In this way, communication requirements are well-defined and no global memory has to be implemented. A process forms a unit of functions and input data which is to be evaluated on another processor.

- Only evaluated values are communicated and therefore we do not need a mechanism that prevents the duplication of work.

- Communication is performed using a one-message protocol in combination with a termination mechanism that could prevent the continued spontaneous transmission of unwanted messages. Although not every communication operation has to be triggered by a specific request, parallel computation and communication are still under the control of demand. Note that the intuition behind this protocol is to "make the common case (i.e. the case that the data is indeed used) fast".

  It goes without saying that termination problems can result from all ways to change the order of evaluation. Especially, if normal form evaluation is introduced using evaluation strategies[THLP98], termination problems can arise that require the programmer to analyze the program more carefully. It has to be stressed that Eden tries to eliminate unneeded computations as far as possible and that the amount of speculative work is much lower than with full normal form evaluation.

  The convention to automatically create demand for top-level process instantiations and for outputs in Eden is a powerful mechanism for the control of demand.

11.3 Implementation of Eden

Prototype implementation with static process networks. The prototype we have presented allows the runtime behaviour of Eden programs to be simulated. In this way, (nearly) the same effects are achieved using different means. Runtime experiments with the MPI+Haskell prototype have shown that the notion of a process introduced in Eden is desirable from an implementation point of view.

This prototype is valuable not only for the implementation of Eden, but also for the further development of the approach, because it makes additional aspects explicit. In this way, it forms a testbed for mechanisms that optimize the topology or granularity of parallel systems.

Design of abstract machine and runtime system. We have presented the abstract machine DREAM, which forms an extension of the well-known Spineless Tagless G-Machine (STG machine). The main benefits of this machine design are: it is orthogonal to the sequential design, so that considerable parts of the sequential implementation can be reused.

11.4 Future work

A framework for explicit parallel programming. In the prototype presented here, placement and granularity are "extremely" explicit - and in this way, it defines a "limit" for parallel functional programming. It would be valuable to establish a detailed classification of several aspects that can be explicit or implicit and testing by practical experiments the impact of making them explicit or not. Based on a corresponding hierarchy of languages that require the explicit description of a different number of aspects, one could start programming with the least explicit version and gradually descend to more explicit versions, if required due to performance issues.

Along these lines, a detailed comparative case study could investigate the relationship between evaluation strategies[THLP98] and Eden, resulting perhaps in possible extensions to both.

Large-scale applications. Part III of this thesis has presented starting points for prototyping and programming large real-world applications in Eden.

The role of nondeterminism. In this thesis, we have used a syntactic criterion for classifying processes as nondeterministic. This pessimistic estimation is satisfactory for the application of equational reasoning, provided that only a small number of Eden programs will make use of nondeterministic features. It would however be an interesting area for future work to investigate how this classification could be made more precise and what programs benefit from nondeterminism.

Bibliography

[And91] Gregory R. Andrews. *Concurrent programming - principles and practice.* Benjamin Cummings, Redwood City, Calif., 1991.

[Ang96] Chris Angus. Numerical Software Development with Functional Languages. In *International Workshop on Modern Software Tools for Scientific Computing (SciTools).* Birkhauser, 1996. also available in electronic proceedings: http://www.oslo.sintef.no/SciTools96/.

[AO91] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language.* Benjamin Cummings, Redwood City, Calif., 1991.

[Aßm96] Claus Aßmann. Coordinating Functional Processes using Petri Nets. In W. Kluge, editor, *Implementation of Functional Languages, Bonn 1996*, LNCS 1268. Springer, 1996.

[AWV96] J.L. Armstrong, M.C. Williams, and S.R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[Bas97] Achim Basermann. Conjugate Gradient and Lanczos Methods for Sparse Matrices on Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 45, 1997.

[BCH+93] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993.

[BCH+94] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[BDS93] Manfred Broy, Claus Dendorfer, and Ketil Stølen. HOPSA - A High-level Programming Language for Parallel Computations. In Peter Spies, editor, *Euro - ARCH*, Springer Verlag, 1993.

[Ber95] B. Berthomieu. Process calculi at work — an account of the LCS project. In *PSLS*, volume 1068 of *Springer LNCS*, 1995.

[Ber97] Bernard Berthomieu. CCS programming in an ML framework: An account of LCS. In Flemming Nielson, editor, *ML with Concurrency – Design, Analysis, Implementation and Application*, Monographs in Computer Science. Springer, 1997.

[BJdM97] Richard Bird, Geraint Jones, and Oege de Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):1–21, sept 1997.

[BK95] George Horatiu Botorog and Herbert Kuchen. Algorithmic Skeletons for Adaptive Multigrid Methods. In *Irregular*, LNCS 980. Springer, 1995.

[BKL97a] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. An Implementation of Eden on Top of Concurrent Haskell. In W. Kluge, editor, *Implementation of Functional Languages, Bonn 1996*, LNCS 1268. Springer, 1997.

[BKL97b] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Functional Programming*, 1997. revised version 1998.

[BKL98a] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. From (sequential) Haskell to (parallel) Eden: An Implementation Point of View. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP)*, LNCS 1490. Springer, 1998.

[BKL+98b] Silvia Breitinger, Ulrike Klusik, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. DREAM - the DistRibuted Eden Abstract Machine. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *Symposium on the Implementation of Funct. Lang. 1997, St. Andrews, selected papers*, LNCS 1467. Springer, 1998.

[BL98] Silvia Breitinger and Rita Loogen. Explicit Process Systems in Eden. In *Constructive Methods for Parallel Programming (CMPP), Marstrand, Sweden*, 1998.

[Ble96a] Guy E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM March 1996/Vol. 39, No. 3*, pages 85–97, 1996.

[Ble96b] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[BLO94] Karsten Bohlmann, Rita Loogen, and Yolanda Ortega-Mallén. Towards a functional process calculus. In *Proceedings of GULP-PRODE, Universita Politecnica de Valencia, Spain*, pages 234 – 250, 1994.

[BLO95] Silvia Breitinger, Rita Loogen, and Yolanda Ortega-Mallén. Concurrency in Functional and Logic Languages. In *Proceedings of the Fuji Int. Workshop on Functional and Logic Programming, Japan*. World Scientific Publishing Company, 1995. ISBN 981-02-2437-0.

[BLO96] Silvia Breitinger, Rita Loogen, and Yolanda Ortega-Mallén. Towards a declarative language for concurrent and parallel programming. In David N. Turner, editor, *Functional Programming, Glasgow 1995*. Springer, 1996.

[BLOP96a] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.

[BLOP96b] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Eden, the Paradise of Functional Concurrent Programming. In *European Conference on Parallel Processing (Euro-Par)*, LNCS 1123. Springer, 1996.

[BLOP97a] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*. IEEE Press, 1997.

[BLOP97b] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. High-level Parallel and Concurrent Programming in Eden. In *APPIA-GULP-PRODE Joint Conference on Declarative Programming, Grado (Italy)*, pages 213–224, 1997.

[BLP98] Silvia Breitinger, Rita Loogen, and Steffen Priebe. Parallel Programming with Haskell and MPI. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Implementation of Functional Languages, London 1998*. University College London, 1998.

[Bot98] George Horatiu Botorog. *High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms*. PhD thesis, RWTH Aachen, Fachgruppe Informatik, 1998. Aachener Informatik – Bericht 97 - 15.

[Brä89] Thomas Bräunl. Structured SIMD Programming in Parallaxis. *Structured Programming*, 10(3):121–132, 1989. Published by Springer, New York.

[Bra95] Lee Braine. The Importance of Being Lazy, 1995. Internal Report, available from http://www.cs.ucl.ac.uk/staff/L.Braine/lazy.html; University College London.

[BW88] R. Bird and Ph. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[Car93] D.B. Carpenter. Some Lattice-based Scientific Programs, Expressed in Haskell. In *FAST: Functional Programming for Arrays of Transputers*. University of Southampton (Report CSTR 93-15) and Imperial College of Science, Technology and Medicine, University of London (Report DOC 93/47), 1993.

[CGKL94] Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, and Hendrik C.R. Lock. Two limits of purely functional parallel programming and how to overcome them. *internal report*, 1994.

[CGKL98] Manuel M. T. Chakravarty, Yike Guo, Margin Köhler, and Hendrik C. R. Lock. GOFFIN: Higher-order functions meet concurrent constraints. *Science of Computer Programming*, 30(1–2):157–199, 1998.

[Cha95] Manuel M.T. Chakravarty. Integrating Multithreading into the Spineless Tagless G-machine. In D. N. Turner, editor, *Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer Verlag, 1995.

[CK94] Daniel J. Clancy and Benjamin Kuipers. Model decomposition and simulation. In *Eighth International Workshop on Qualitative Reasoning about Physical Systems (QR-94)*, 1994.

[Col89] M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations.* MIT Press, 1989.

[CS97] David Culler and Jaswinder Pal Singh. *Parallel Computer Architecture – A Hardware / Software Approach.* Morgan Kaufmann, 1997.

[DAB+95] Jack Dennis, Shail Aditya, Wim Böhm, Chris Kirkham, and James McGraw. Panel: Nondeterminacy in Functional Programming: An Essential Feature or a Programmer's Nightmare? In *High Performance Functional Computing*, 1995.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *Parallel Architectures and Languages Europe.* Springer, 1993.

[DGTY95] J. Darlington, Y.-K. Guo, H.W. To, and J. Yang. Functional skeletons for parallel coordination. In *Euro-Par*, LNCS 966. Springer, 1995.

[dKC94] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295 – 336, 1994.

[DR83] I. 5. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[(ed95] B. Szymanski (ed.), editor. *Integrating Data and Task Parallelism in Scientific Programs.* Kluwer, 1995.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, 1997.

[EL98] Nils Ellmenreich and Christian Lengauer. On Functional Scientific Computing. In *Glasgow Workshop on Funct. Prg. 1997*, 1998. submitted for publication.

[EPZ95] R. Ebner, A. Pfaffinger, and C. Zenger. FASAN – eine funktionale Agenten-Sprache zur Parallelisierung von Algorithmen in der Numerik. In W. Mackens and S. Rump, editors, *Software Engineering in Scientific Computing.* Vieweg Verlag, 1995. in German.

[FHJ92] Torkel Franzén, Seif Haridi, and Sverker Janson. An overview of the Andorra Kernel Language. Technical report, Swedish Institute of Computer Science, 1992.

[Fis95] Paul A. Fishwick. *Simulation Model Design and Execution.* Prentice Hall, 1995.

[Fok95] Jeroen Fokker. Functional Specification of the JPEG algorithm and an Implementation for Free. In R.C. Veltkamp and E. H. Blake, editors, *Programming Paradigms in Graphics – Eurographics workshop, Maastricht.* Springer, 1995.

[Fre96] Vincent W. Freeh. A Comparison of Implicit and Explicit Parallel Programming. *Journal of Parallel and Distributed Computing*, 34:50–65, 1996.

[FT90] Ian Foster and Stephen Taylor. *Strand – New Concepts in Parallel Programming.* Prentice Hall, 1990.

[GBDJ94] Al Geist, Adam Beguelin, Jack Dongarra, and Weicheng Jiang. *PVM: Parallel Virtual Machine.* MIT Press, 1994.

[GC92] David Gelernter and Nicolas Carriero. Coordination languages and their significance. *Comm. of the ACM*, 35(2), 1992.

[GGHL+96] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdane, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing Interface. In *European Conference on Parallel Processing (Euro-Par)*, LNCS 1123, pages 128 – 135. Ecole Normale Supérieure de Lyon, Springer Verlag, 1996.

[GH96] Thomas Gehrke and Michaela Huhn. ProFun - A Language for Executable Specifications. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP)*, LNCS 1140, pages 304 – 318. Springer, 1996.

[GHW90] H. Glaser, P. Hartel, and J. Wild. A pragmatic approach to the analysis and compilation of lazy functional languages. In *Proceedings of the 2nd Conf. on Parallel and Distributed Processing*, pages 169–184. North Holland, 1990.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* MIT Press, 1994.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *Journal of Parallel Prog.*, 18(2), 1989.

[GMP94] A. Giacalone, P. Mishra, and S. Prasad. Facile Chemistry Revised. Technical Report 94-36, ECRC, 1994.

[GPP96] Luis A. Galán, Cristóbal Pareja, and Ricardo Peña. Functional Skeletons Generate Process Topologies in Eden. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP)*, LNCS 1140. Springer, 1996.

[GSWZ95] P. W. Grant, J. A. Sharp, M. F. Webster, and X. Zhang. Experiences of parallelising finite-element problems in a functional style. *Software - Practice and Experience*, 25(9):947–974, September 1995.

[GSWZ96] P. W. Grant, J. A. Sharp, M. Webster, and X. Zhang. Sparse matrix representations in a functional language. *Journal of Functional Programming*, 6(1):143–170, January 1996.

[GTDoCS98] University of Glasgow GHC Team Department of Computing Science. The Glasgow Haskell Compiler User's Guide, version 3.02, 1998.

[Hai94] Gaétan Hains. Parallel functional languages should be strict. In Pehrson and Simon, editors, *Workshop on GPPP, World Computer Congress*, volume 1, pages 527–532, Hamburg, September 1994. IFIP, North-Holland.

[Ham94] Kevin Hammond. Parallel Functional Programming: an Introduction. In *PASCO, Linz, Austria*. World Scientific, 1994.

[Ham97] Kevin Hammond. Parallel Cost-Centre Profiling. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *Symposium on the Implem. of Funct. Lang., St. Andrews*, 1997.

[Har96] Dave A. Harrison. Process Oriented Discrete Event Simulation Using Monads. Technical Report NPC-TRS-96-1, University of Northumbria at Newcastle - Department of Computing, 1996.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with fudgets. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 137–182. Springer, 1995.

[HFA+96] Pieter H. Hartel, Marc Feely, Martin Alt, Lennart Agustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. Van Groningen, Kevin Hammond, Bogumił Hausman, Melody Y. Ivory, Richard E. Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D. Lins, Sandra Loosemore, Niklas Röjemo, Manuel Serrano, Jean-Pierre Talpin, Jon Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and Peter Wentworth. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6(4), 1996.

[Hin92] Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda*. Teubner Verlag, 1992.

[HLT+97] Cordelia Hall, Hans-Wolfgang Loidl, Phil Trinder, Kevin Hammond, and John O'Donnell. Refining a Parallel Algorithm For Calculating Bowings. In *Glasgow Workshop on Funct. Prg.*, 1997.

[Hor96] Matthias Horn. A Different Integration of Multithreading into the STGM. In W. Kluge, editor, *Impl. of Funct. Lang.* Kiel University, 1996.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, 1996.

[Hud91] Paul Hudak. Para-Functional Programming in Haskell. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press, 1991.

[HXA96] Kai Hwang, Zhiwei Xu, and Masahiro Arakawa. Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):522–535, May 1996.

[JH93] Mark P. Jones and Paul Hudak. Implicit and Explicit parallel Programming in Haskell. Technical Report RR-982, Yale University, 1993.

[JS98a] C.B. Jay and P.A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98 Held as part of the joint european conferences on theory and practice of software, ETAPS'98 Lisbon, Portugal, March/April 1998*, LNCS 1381, pages 139–53. Springer, 1998.

[JS98b] C.B. Jay and P.A. Steckler. Polymorphism over nested regular arrays: theory and implementation in fish. Technical Report 01, Sydney University of Technology, 1998. To appear in FMCS'98.

[Kel89] Paul Kelly. *Functional Programming for Loosely Coupled Multiprocessors.* Pitman, 1989.

[Kes96] Marco Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, Katholieke Universiteit Nijmegen, 1996.

[KM77] G. Kahn and D. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *IFIP Congress on Information Processing.* North-Holland, 1977.

[Kuc96] Herbert Kuchen. Eine datenparallele funktionale Sprache für Rechner mit verteiltem Speicher. In W. Mackens and S. Rump, editors, *Software Engineering in Scientific Computing.* Vieweg Verlag, 1996. in German.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages (POPL) 1993*, ACM Press, pages 144–154, 1993.

[LH94] Ben Lee and A. R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, 27(8):27–39, 1994.

[LH97] F. Loulergue and G. Hains. Functional Parallel Programming with explicit processes: beyond SPMD. In *European Conference on Parallel Processing (Euro-Par)*, LNCS. Springer Verlag, 1997.

[LHP94] Hans Wolfgang Loidl, Kevin Hammond, and Andrew Partridge. Solving Systems of Linear Equations Functionally: a Case Study in Parallelisation. Technical report, University of Glasgow, 1994.

[LJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 24–35, Orlando, Florida, 1994.

[LKC93] Junxian Liu, Paul Kelly, and Stuart Cox. Functional Programming for Finite Element Analysis. In *FAST: Functional Programming for Arrays of Transputers*. University of Southampton (Report CSTR 93-15) and Imperial College of Science, Technology and Medicine, University of London (Report DOC 93/47), 1993.

[Loi97a] Hans Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, 1997. TR-1998-7.

[Loi97b] Hans Wolfgang Loidl. LinSolv: A Case Study in Strategic Parallelism. In *Glasgow Workshop on Funct. Prg.*, 1997.

[LW92] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In David Abramson and Jean-Luc Gaudiot, editors, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, 1992. ACM Press.

[Mic96] O. Michel. Introducing dynamicity in the data-parallel language 81/2. In *European Conference on Parallel Processing (Euro-Par)*, LNCS 1123. Springer, 1996.

[Moh97] Markus Mohnen. *Optimizing the Memory Management of Higher-Order Functional Programs*. PhD thesis, RWTH Aachen, Fachgruppe Informatik, 1997. Aachener Informatik – Bericht 97 - 13.

[MPI94] The MPI Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, 1994.

[MPI97] The MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, July 1997.

[MW92] José Meseguer and Timothy Winkler. Parallel Programming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, pages 253–293. Springer, 1992.

[NAH+95] R.S. Nikhil, Arvind, J. Hicks, Sh. Aditya, L. Augustsson, J.-W. Maessen, and Y. Zhou. pH Language Reference Manual, Version 1.0 - preliminary. Technical report, Computation Structures Group Memo 369, Laboratory for Computer Science, MIT, 1995.

[NP96] Thomas Nordin and Simon Peyton Jones. Green Card: A foreign-language interface for Haskell. Technical report, Department of Computer Science, University of Glasgow, 1996.

[NSEP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer. Concurrent Clean. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *LNCS*, pages 202–219. Springer, 1991.

[OR97] John O'Donnell and Gudula Rünger. A Coordination Level Functional Implementation of the Hierarchical Radiosity Algorithm. In *Glasgow Workshop on Funct. Prg. 1997*, 1997.

[OW98] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 8(1), January 1998.

[Pan96] Prakash Panangaden. Does Concurrency Theory Have Anything To Say About Parallel Programming. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 58:140 – 147, Feb. 1996.

[Pey92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. of Funct. Prog.*, 2(2), 1992.

[PGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjørn Finne. Concurrent Haskell. In *ACM Symp. on Principles of Progr. Lang. (POPL)*, 1996.

[PH97] John Peterson and Kevin Hammond (eds.). Report on the programming language Haskell: a non-strict, purely functional language, version 1.4. Technical Report YALEU/DCS/RR-1106, Yale University, 1997.

[Pie94] Benjamin C. Pierce. PICT: An experiment in concurrent language design. Pict version 3.6 tutorial, University of Edinburgh, March 1994.

[PJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. Technical report, Department of Computer Science, University of Glasgow, 1997.

[PL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

[Plü97] Mari Plümacher. From PVM to MPI: A Case Study. Diplomarbeit. Philipps-Universität Marburg, 1997. in German.

[Poh95] Werner Pohlmann. Towards functional simulation programming. *SAMS*, 20:187–207, 1995.

[PR97] Prakash Panangaden and John Reppy. The Essence of Concurrent ML. In Flemming Nielson, editor, *ML with Concurrency – Design, Analysis, Implementation and Application*, Monographs in Computer Science. Springer, 1997.

[Pri97] Steffen Priebe. Parallele funktionale Programmierung mit Haskell und MPI. Fortgeschrittenenpraktikum. Philipps-Universität Marburg, 1997. in German.

[PS97] Peter Pepper and Mario Südholt. Deriving parallel numerical algorithms using data distribution algebras: Wang's algorithm. In *Proceedings of the 30th Hawaii International Conference on System Sciences*. IEEE, January 1997.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 1997.

[Que92] Christian Queinnec. A concurrent and distributed extension to scheme. In D. Etiemble and J-C. Syre, editors, *parle92*, LNCS 605. Springer, 1992.

[Qui94] M. J. Quinn. *Parallel Computing—Theory and Practice.* Mc.Graw Hill, 1994.

[Re94] Colin Runciman and David Wakeling (eds.). *Functional Languages Applied to Realistic Exemplars: the FLARE Project.* UCL Press, 1994.

[Rep91] J. Reppy. CML: A higher-order concurrent language. In *ACM PLDI*, 1991.

[Saa96] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing, 1996.

[San95] André Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Glasgow University, Department of Computing Science, 1995.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[SB90] J. Stoer and R. Bulirsch. *Einfuehrung in die Numerische Mathematik II (3. Auflage).* Springer, Berlin, 1990. in German.

[Sch94] Michael Schiek. Quantifizierung und Modellierung der respiratorischen Sinusarrhythmie. Technical Report 2899, Forschungszentrum Jülich GmbH, 1994. in German.

[Sch96a] Enno Scholz. PIDGETS — unifying pictures and widgets in a constraint-based framework for concurrent functional GUI programming. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP)*, LNCS 1140. Springer, 1996.

[Sch96b] Sven-Bodo Scholz. *Single Assignment C – Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen.* PhD thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, 1996. in German.

[Sch97a] Sven-Bodo Scholz. On Programming Scientific Applications in SAC - A Functional Language Extended by a Subsystem for High-Level Array Operations. In W. Kluge, editor, *Implementation of Functional Languages, Bonn 1996*, LNCS 1268. Springer, 1997.

[Sch97b] Andreas Schwarzer. *Skalierungstechniken für k-bestimmte Verzweigungsprobleme und Iterationsmethoden für große, dünn besetzte Eigenwertprobleme.* PhD thesis, Philipps-Universität Marburg, Fachbereich Mathematik, 1997. in German.

[Sch98] Gaby Schulemann. Report Supercomputer: Richtige Rechner – Supercomputer auf dem Weg zum Weltsimulator. *c't Magazin für Computertechnik*, 11:82–92, 1998. in German.

[SDA95] M. Schiek, F.R. Drepper, and H.-H. Abel. Mathematisches Modell der 'respiratorischen' Sinusarrhythmie. *Wien. med. Wschr.*, 145:492 – 494, 1995. in German.

[Ser97a] Pascal R. Serrarens. Distributed Arrays in the Functional Language Concurrent Clean. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *European Conference on Parallel Processing (Euro-Par)*, LNCS 1300. Springer, 1997.

[Ser97b] Pascal R. Serrarens. Implementing the Conjugate Gradient Algorithm in a Functional Language. In W. Kluge, editor, *Implementation of Functional Languages, Bonn 1996*, LNCS 1268. Springer, 1997.

[Sha89] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3), 1989.

[Ske91] Stephen K. Skedzielewski. Sisal. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press, 1991.

[Sla96] Michael Slater. The microprocessor today. *IEEE Micro*, December 1996.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*, LNCS 1000. Springer, Berlin, 1995.

[SPOP97] M. Südholt, C. Piepenbrock, K. Obermayer, and P. Pepper. Solving large systems of differential equations using covers and skeletons. In *50th IFIP WG 2.1 Working Conference on Algorithmic Languages and Calculi*. Chapman & Hall, Feb 1997.

[SSV96] W. Schulte, T. Schwinn, and T. Vullinghs. TkGofer: A functional GUI library. 1996.

[Sto89] J. Stoer. *Numerische Mathematik 1 (5. Auflage)*. Springer-Verlag, Berlin, 1989. in German.

[Sud96] Katrin Suder. Nichtlineare Analyse eines Modells der respiratorischen Sinusarrhythmie. Technical Report 3213, Forschungszentrum Jülich GmbH, 1996. in German.

[THL+97] P. W. Trinder, K. Hammond, H. W. Loidl, S. L. Peyton Jones, and J. Wu. Go-faster Haskell Or: Data intensive Programming in Parallel Haskell. In *Glasgow Workshop on Funct. Prg.*, 1997.

[THLP96] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. In Werner Kluge, editor, *Workshop on the Implementation of Functional Languages, Bonn, Germany*. University Kiel, 1996.

[THLP98] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.

[THM+96] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *ACM PLDI*, 1996.

[Tho93] John Thornley. Integrating functional and imperative parallel programming: CC++ solutions to the salishan problems. Technical Report CS-TR-93-40, California Institute of Technology, 1993.

[Tho94] B. Thomsen. Polymorphic Sorts and Types for Concurrent Functional Programs. In *6th int. Workshop on the Implementation of Functional Languages*, University of East Anglia, Norwich, 1994.

[TKB92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, 1992.

[TLK97] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. FACILE – From Toy to Tool. In Flemming Nielson, editor, *ML with Concurrency – Design, Analysis, Implementation and Application*, Monographs in Computer Science. Springer, 1997.

[Tur92] David A. Turner. An Approach to Functional Operating Systems. In David A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1992.

[Val90] L. G. Valiant. A bridging model for parallel computations. *Communications of the ACM*, 33(8):103–111, 1990.

[vENPS89] 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.

[vEPS89] M. C. J. D. van Eekelen, M. J. Plasmeijer, and J. E. W. Smetsers. Communicating functional processes. Technical Report 89-3, University of Nijmegen, 1989.

[vG97] John H.G. van Groningen. The Implementation and Efficiency of Arrays in Clean 1.1. In W. Kluge, editor, *Implementation of Functional Languages, Bonn 1996*, LNCS 1268. Springer, 1997.

[Wad96] Philip Wadler. Lazy versus strict. *ACM Computing Surveys*, 28(2):318–320, June 1996.

[Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[Wat96] Andrew Watson. Corba fundamentals. Technical report, Object Management Group, 1996.

[Wen98] Lothar Wenzel. Nah am Original – Wie arbeiten Simulatoren? *c't Magazin für Computertechnik*, 4:214–223, 1998. in German.

[Wik94] Claes Wikström. Distributed programming in Erlang. In Hoon Hong, editor, *PASCO'94: First International Symposium on Parallel Symbolic Computation*, pages 412–421. World Scientific Publishing Company, 1994.

[Wis92] David S. Wise. Matrix algorithms using quadtrees. Technical Report 357, Indiana University, Computer Science Department, 1992.

[WS92] Robert L. Wainwright and Marian E. Sexton. A study of sparse matrix representations for solving linear systems in a functional language. *Journal of Functional Programming*, 2(1):61–72, January 1992.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Frontier Series. Addison-Wesley, 1991.

Lebenslauf

Silvia Breitinger

21.05.1969	geboren in München als einzige Tochter von Heinz und Roswitha Breitinger, geb. Schulze
Juni 1988	Abitur am Gymnasium Geretsried mit Note „Sehr gut“ (1,1)
Nov. 1988 - Nov. 1993	Studium der Informatik an der Technischen Universität München: Diplom mit Note „Gut“ (1,7)
	Förderung durch ein Stipendium des Stifterverbandes für die deutsche Wissenschaft (IBM - Fonds)
Nov. 1992 - Mai 1993	Diplomarbeit im Wissenschaftlichen Zentrum der IBM in Heidelberg
	Titel: Untersuchung des Einsatzes von Constraint - Handling - Verfahren im Bereich Produktionsplanung
Dez. 1993	Einstellung als wissenschaftliche Mitarbeiterin an der Philipps - Universität Marburg im Fachbereich Mathematik und Informatik, Fachgebiet Informatik