Structured Generic Programming in Eden

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

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

Steffen Priebe
aus Marburg/Lahn

Marburg/Lahn 2007

Vom Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg als Dissertation am 3. Februar 2007 angenommen.

Erstgutachterin: Prof. Dr. Rita Loogen
Zweitgutachter: Hochschuldozent Dr. Ralf Hinze, Universität Bonn

Tag der mündlichen Prüfung am 9. Februar 2007

Zusammenfassung

Die Ausnutzung von Parallelität ist schon immer eine für den Benutzer unsichtbare, aber große Quelle zur Verbesserung der Prozessorleistung gewesen. Da aber die Menge an implizit nutzbarer Parallelität auf Befehlsebene begrenzt ist und gleichzeitig immer größere Rechenleistungen benötigt werden, erleben wir gegenwärtig eine Renaissance expliziter paralleler Techniken sowohl auf Hardware- als auch auf Softwareebene. Aufgrund ihrer Komplexität sind parallele Systeme mit traditionellen Programmiersprachen schwer zu handhaben. Deshalb werden zunehmend abstraktere Programmiersprachen betrachtet. *Eden* ist eine solche Sprache, die Programmkonstrukte zur gleichzeitigen Auswertung von Ausdrücken in die funktionale Programmiersprache Haskell integriert. Eden ist ein Kompromiss zwischen vollständiger und fehlender Parallelitätskontrolle durch den Programmierer: Es wird genügend Kontrolle zur Erreichung guter Beschleunigungen bereitgestellt, gleichzeitig wird aber auch ein abstrakter Programmierstil durch die Übernahme weniger wichtiger Vorgänge durch das Laufzeitsystem gewahrt.

In dieser Arbeit stellen wir Eden drei Sprachkonzepte zur Seite, um eine noch weitergehende Abstraktion zu erreichen:

- Unter *Meta-Programmierung* versteht man die Definition von Programmen, die andere Programme erzeugen oder verändern. Diese Technik wird benutzt, um in Haskell programmierte statische Präprozessorphasen zur Verbesserung von Eden-Programmen zu konstruieren. Dadurch wird die Portabilität des Eden-Compilers verbessert, da diese Phasen nun nicht mehr in den zugrundeliegenden Haskell-Compiler integriert werden müssen.

- *Generische Programmierung* erweitert parametrische zur strukturellen Polymorphie und ermöglicht daher die Definition von Funktionen, die mit beliebigen Argumentdatenstrukturen funktionieren. Wir beschreiben einen reduzierten struktur-orientierten Ansatz zur generischen Programmierung, der für den Einsatz in Eden konzipiert wurde. Mit diesem Ansatz definieren wir allgemeine parallele Verarbeitungsschemata.

- Möglichkeiten zur *Auswertungskontrolle* müssen vorhanden sein, wenn in einer funktionalen Sprache Bedarfssteuerung auf Parallelität trifft. Deren Ziele sind gegensätzlich: Auf der einen Seite werden Auswertungen zeitlich nach hinten geschoben, während auf der anderen Seite frühe Auswertung die Gleichzeitigkeit und damit die Parallelität fördert. Wir zeigen Wege auf, um die Auswertung zu Gunsten der Parallelität zu steuern.

In funktionalen Sprachen wird der Auswertungsverlauf durch Datenabhängigkeiten sowie durch Kontrollkonstrukte bestimmt. Analog kann man bei parallelen funktionalen Programmen zwischen *daten-orientierten* und *kontroll-orientierten* unterscheiden. Entsprechend zeigen wir zunächst generische Methoden zur Partitionierung von Datenstrukturen sowie generische Versionen der parallelen map-Funktion. Dann stellen wir kontroll-parallele Methoden vor, mit denen die in Eden oft vorkommenden Datenströme behandelt werden können; zusätzlich zeigen wir parallele Schemata zur effizienten Verarbeitung irregulärer Strukturen und langer Kommunikationswege. Letztendlich vereinigen wir die gezeigten Techniken in einer Programmentwicklungsmethodik für Eden.

Abstract

Parallelism has always been a hidden main source of processor power. As a result of the limited amount of implicitly exploitable small-scale parallelism (for example on the instruction-level) and ever-growing needs for more computational power, parallel techniques break their way from a minor matter to a major feature in both hardware and software. Due to their complexity, such parallel systems are getting increasingly difficult to control with conventional programming languages. Therefore, more abstract high-level approaches move into focus. *Eden* is a representative of these approaches which integrates constructs for remote evaluation into the standard functional language Haskell. It strikes a balance between full and no parallelism control and delivers good speedups while providing a high-level style of programming.

In this thesis we equip Eden with three language features to raise the abstraction level even more:

- *Meta-programming*, which means that programs manipulate other programs, will be used to define static preprocessing steps coded in Haskell for enhancing Eden programs. This supports portability of the Eden compiler, as some transformations can be pulled out of the foreign Haskell implementation.

- *Generic programming* raises parametric to structural polymorphism and allows to write functions which are valid for all data structures. We will present a reduced, structure-oriented approach to generic programming tailored for Eden's needs. Using this approach, very general parallel schemes are defined.

- *Demand control* is a basic requirement if a lazy functional language is faced with parallelism. The contradictory aims of postponing evaluations and simultaneity of evaluations enforces demand control in favour of parallelism. We present a set of means to do that.

In functional programs, evaluation progress is determined by a mix of control structures and data dependencies. Accordingly, parallel functional programs can roughly be classified into *data-oriented* and *control-oriented* ones. Firstly, we will present generic methods for partitioning data structures as well as generic versions of the parallel map function. Secondly, we will show methods to manage the omnipresent streams as well as parallel schemes for dealing with irregular task sizes and long communication distances. To conclude, we will summarise all methods shown in a program developing guide for Eden.

Acknowledgements

My thanks are threefold:

Firstly, I want to most heartily thank my supervisor Rita Loogen. Not only for introducing me into the exciting topics of functional languages and parallel computers, but also for giving me the possibility to write this thesis about the topics I am most interested in and for providing a very stimulating working atmosphere. Thanks also go to the many present and past members of the Eden group in Marburg and Madrid for many valuable discussions. I am grateful to Ralf Hinze for agreeing to serve on the examination board despite being loaded with teaching duties and research activities. I am also thankful for the suggestions of other conference participants and especially for the neutral assessments and helpful comments of many anonymous referees.

Secondly, I am most thankful to the Evangelisches Studienwerk Villigst e.V. (which is the scholarship organisation of the german protestant church) for granting me a scholarship when I needed it the most. Not only their generous financial support is acknowledged, but at least as much the inspiring, interdisciplinary meetings at Villigst and in the local group of scholarship holders at Marburg.

Thirdly, the warmest thanks go to my family for their constant and loving support in all situations. Above all, my loving thanks go to my wife Friederike and our daughter Luisa for bearing with me through all ups and downs of my work.

Contents

List of Figures

1. Introduction

"More positively, once the programmer has dreamt up a parallel algorithm, they want to be able to express that parallelism in an explicit way in the program. Writing a vanilla functional program, and hoping that a cunning compiler will be able to figure out your intentions, is not good enough ..."

"Second, the always dubious 'feed in an arbitrary program and watch it run faster' story is comprehensively dead."

Simon Peyton Jones[80]

1.1 Motivation

The fundamental motivation for starting this thesis grew out of the fascination for a blend of two exciting, yet often considered peripheral, fields of research in computer science. The combination of their often contradictory characteristics has already attracted lots of research, but a standard and commonly acknowledged solution has not yet been achieved. This thesis describes another way of *functionally programming parallel computers*.

1.1.1 Why Parallel Computing Is on the Rise Again

Forty years ago "it seemed that the future of computers was parallel", as everyone was working on "multiprocessors, vector processors, array processors, dataflow-driven pipelined processors" (Michael J. Flynn in the preface of the Parallel and Distributed Computing Handbook[217]). At that time, Flynn proposed his nowadays famous parallel computer taxonomy[62] based on having either a single or multiple instruction and data streams. Then, rather unexpected, remarkable advances in uniprocessor and memory technology were made. The uniprocessor price-to-performance ratio became unbeatable and parallelism took a back seat for some time, although a few high-performance supercomputers have always been present.

Today, we experience a renaissance of parallel computing, which has often been predicted for the future and now has become reality. Parallel technologies (like pipelining, instruction-level parallelism via superscalar processors, and Very Long Instruction Words[48]) have also been used in the past, but more as implicit means

for performance-boosting in the background. But these seem to have found their limits, as the lookahead for exploiting instruction-level parallelism and chip die sizes housing ever-growing caches have to be extended more and more; additionally, since 2003 CPU clock frequencies are hardly advancing anymore, and the 4 GHz barrier seems to be a tough one. Therefore uniprocessor development is heading more towards revealing explicit parallelism since a couple of years to gain performance. There are not only massively parallel GPUs for graphics processing (which are also bundled as symmetric multiprocessing nodes), but also multi-core processors which climb the step from thread-level (for example hyperthreading) to chip-level parallelism. With the success of the Internet, also new network-based techniques like distributed computing using idle resources and Grid computing [49] are on the rise.

1.1.2 Why More Abstract Programming Languages Are Needed

Together with the change in everyday systems also the bigger classical parallel systems change: Nodes are getting faster and are equipped with ample storage, network bandwidths are also increasing steadily. All this is very desirable, but the situation is not all roses from a programming point of view as three fundamental problems in exploiting this huge computational power persist [217] :

- *Finding large degrees of parallelism:* It is getting harder to generate or find enough parallelism to adequately employ parallel nodes. Often traditional sequential algorithms have to be reconsidered, as they do not expose enough parallelism or lead into false directions of parallelisation.
- *Efficiently executing that parallelism:* Speedup, which is the performance advantage gained from using multiple nodes, has to be realised despite substantial system overheads introduced by managing parallel execution. A parallel runtime system has to care for scheduling, communication, synchronisation, and memory consistency.
- *Efficiently expressing that parallelism in a language:* Cooperation and coordination has to be expressible in an abstract and concise way. This is most important for later modifications, reuse, reasoning, and comprehensibility.

While the first problem is one of finding new parallelisable algorithms the second one is about implementing efficient compilers. In the following, we focus on solving the third problem.

Originating from a quite technical direction, parallel computers first attracted practitioners introducing parallelism into technical low-level languages. Conventional languages with parallel extensions tried to swallow parallelism as another side feature or library, while not treating it as a substantial part of the language. Indeed, effective parallel programming is possible with these languages, but at the cost of quickly getting extremely complex and error-prone programs hard to reason about. Still it is common to run the most advanced parallel computers with programs written in languages based on the most low-level approaches (for example C [115] and MPI [76]).

For expressing parallelism and defining the way of internode cooperation, many different more abstract approaches have appeared; but an easy and commonly applicable programming model is still missing. Automatic parallelisation of traditional programs has proven to be not successful, and parallelism has found its way into programming languages being not a background feature but a prominent and visible one. Due to the complexity of managing parallel activities (cooperating to solve problem while competing for shared resources like network bandwidth, processor time, and memory access), one can get full parallelism control in a simple low-level language or little control in a comfortable high-level language, usually not both at the same time.

Due to the complex interplay between parallel architecture, operating system, programming language, compiler, algorithm, and algorithm implementation, monitoring or tracing tools have to be used for both performance tuning and debugging. The wealth of existing tools (Chapter 9 of Foster[64] , Chapter 31 of Zomaya[217]) also indicates a need for more abstract programming approaches, as these tools are often used for real debugging and not just for improving an already good performance.

In total we now have the odd situation, that the available parallel computing power is growing rapidly, while this power cannot always be used efficiently. As Ebcioğlu et al.[59] put it:

> "It is now common wisdom that the ongoing increase in complexity of large-scale parallel systems to address these challenges [increase of node speeds, fast uniform memory access, use all levels of parallelism] has been accompanied by a *decrease in software productivity* for developing, debugging, and maintaining applications ..."

Moreover, parallel computers are often used as a pool of sequential nodes for executing algorithms based on trivial parallelisations where tasks have no or only small dependencies. It causes not too many problems to express these in a low-level language, while more complex problems are much harder to express and to solve. More abstract languages can provide the means for expressing high-level parallel schemes in a concise and provably correct way while providing predictable performance.

Corresponding to the aim of building parallel computers with the highest possible performance, many people design parallel programs aiming only at achieving maximum performance above everything else while sacrificing clarity. Parallel program development is accordingly not as structured as one would wish and is in need of more abstract programming languages and development methods.

1.1.3 Why Functional Programming Languages May Be the Solution

As applications are getting more complex, this is compensated by programming getting more abstract. This is reflected by

- the increasing presence of *sophisticated tools supporting program development*, like integrated development environments (IDEs) or extensive libraries as for example in Java [75] .

- the increasing *abstraction in programming languages*. Object-oriented languages, being the followers of their less abstract imperative predecessors C and Pascal, are extremely popular because of their abstraction and maybe even more because of their extensive standardised libraries. Declarative languages are also attracting more and more interest.

IDEs, extensive libraries, and tools for post-mortem performance analysis alone are useful, but cannot replace a truly abstract parallel language; therefore, the need for such a language which treats parallelism not as an aside but as one of its main features is evident.

Traditional imperative languages tend to be extremely verbose, low-level, and artificially sequential. This makes them not an ideal target for integrating parallel language constructs. But even today the most recent approaches to parallel programming developed by three big supercomputer vendors are based on imperative languages: X10[59] by IBM is closely related to Java, Chapel[36, 38] by Cray Inc. also (but integrates functional features like parametric polymorphism), and Fortress[196] by Sun Microsystems is essentially Fortran.

Functional languages are in contrast not only a radically different approach to programming, but also have a lot of advantageous features. Hughes in his famous paper[105] and Hammond and Michaelson in their comprehensive work [80] identify the following advantages:

- Ease of program construction
- Ease of code reuse
- Simplicity, generality (higher-order functions), and abstraction
- Ease of reasoning
- Ease of program transformation and scope for optimisation

All these advantages carry over from a sequential to a parallel setting. Hammond and Michaelson[82, 80] argue, that functional languages are additionally qualified for integrating parallelism because of:

- Ease of parallel partitioning

  As a functional program lacks implicit control dependencies (such as assignment), only the clearly visible explicit control dependencies and data dependencies limit the parallel evaluation of any pair of subexpressions. Thus functional programs reveal quite directly and naturally all parallelism they contain.

- Data dependencies reveal communication needs

  Communication needs display themselves naturally via data dependencies and provide a fine-grained communication network which only needs to be coarsened to a degree which fits the parallel algorithm and the execution environment.

- Deadlocks are hard to create

  Parallelising a purely functional program will not introduce deadlocks. Only if more powerful constructs are present, deadlocks can occur in rare situations.

- Strong connection to sequential version

  Usually it is easy to move from the parallel to the sequential version and back for testing purposes, as parallel constructs fit nicely into the functional style of expression. In purely functional programs, evaluation order does not matter so that results will always be the same; correspondingly, non-termination will also occur in both settings.

- Natural expression of high-level parallelism constructs

  Due to the higher-order functional style, high-level parallelism schemes can be expressed quite naturally; this is done by a functional input-output description annotated with parallel constructs.

As in every approach to parallel programming, the distribution of parallelism control between the runtime system and the programmer can vary also here. Following the taxonomy of Hammond and Michaelson[80] in their Chapter 1.2.4, the control exerted by the programmer can range from none to all. *Purely implicit* approaches directly derive available fine-grained parallelism from a given sequential program with the help of none or only few annotations. *Restricted implicit* approaches identify more coarse-grained parallelism and exchange for example a higher-order function by a predefined parallel one; thus this approach is restricted to a collection of predefined cases. *Controlled parallelism* approaches rely on explicit parallel annotations by the programmer. Implicit variants hand some control over to the runtime system by allowing the annotation to be ignored; explicit variants are rigorous by executing every annotation. Truly *explicit approaches* reveal every detail of parallel coordination on the program level and let the programmer specify each detail.

Explicit approaches are not desirable, as the complete burden of correctly specifying the parallel collaboration is shifted to the programmer. On the other hand, fully implicit approaches have not proven successful in the past. If a program is written without aiming at a later parallelisation, an automatic system may have a hard time generating an acceptable speedup. And if it is written with a parallelisation in mind, the system is given no hint about the programmer's intentions and may have again problems finding out. This leaves us with two approaches which both leave some control to the programmer and some to the runtime system, each with a bias to one side. This is where the current research mostly focuses at the moment.

In our thesis, we will work with the parallel *Haskell* [164] dialect *Eden* [143] . Eden follows the controlled parallelism branch by introducing explicit process creation constructs with implicit communication and termination. This way, the programmer has enough control for specifying an efficient parallel program while much tedious work is left for the runtime system. We will describe Eden in some detail in Chapter 2.5. Eden inherits a wealth of libraries and tools from Haskell and provides itself a specialised performance monitoring tool.

1.1.4 Why More Advanced Programming Techniques Are Needed

Eden is already a powerful and abstract parallel programming language. Then why do we claim that the current programming techniques are insufficient? We do because currently programming is done mainly by using some basic and well-known sequential methods. Additionally, a couple of non-uniform parallel skeletons have been collected. Program design is then often done by attaching code to a selected skeleton or by inserting parallel constructs into sequential legacy code. Therefore there is a need for a more advanced programming style. We have identified the following extensions from which programming in Eden can benefit:

- *Meta-programming*

  Different language design lines usually each possess one or two standardised and commonly agreed-on representatives. C is a representative for imperative languages, while Java is one for imperative object-oriented languages. Strict functional languages are represented by ML [163] , while the advances in non-strict functional languages are cumulated in Haskell [164] . For each of them, at least one reference implementation exists. For Haskell, this is the Glasgow Haskell Compiler[165] , GHC for short. Apart from standard languages, different domain-specific extensions of these languages (like parallelism for Eden) arise. Understandably these are mostly implemented by extending the corresponding standard implementation. Unfortunately, the base language implementation is progressing rather quickly, forcing the extension to be carried over many different versions to stay comparable with other extensions. This forces the extension implementation to be small and portable. As the base languages are often implemented in their own language via bootstrapping, one could think of a library expressing the extension. But this is often not viable, as extensions may need to change internals (like parallelism needs to change the runtime system). Nevertheless the aim should be to implement domain-specific extension as far as possible as a library with only minimal base compiler modifications to stay portable.

  Meta-programming[189] can help by providing static program code access for implementing domain-specific program analyses and transformations as a library. It can also help specialising the program according to its parallel execution environment, on which parallel program performance depends because of the differences in node number, node architecture, network topology and bandwidths.

- *Demand control*

  Parallelism relies on the strong control of evaluation progress; in contrast, laziness represents loose control of evaluation with unclear evaluation order and without execution guarantees. For Eden to be efficient we use mixed evaluation which needs to combine strong parallel control and lazy sequential control. Although being sufficient for achieving an efficient program, undocumented demand control scattered across a program is not helpful when it comes to later modifications. For clear program structures we need a clear demand control which also separates both concerns.

- *Generic programming and skeletons*

  Depending on the parallel algorithm and design choices, parallelism may either reveal itself dominantly either via *data structures* or via *control structures*.

  Data parallelism can be abstracted via parametric polymorphism and even more via structural polymorphism (generic programming) used for more abstract parallel functions. As nested data structures are omnipresent in parallel programming, the approach to generic programming has to provide general and stepwise access to each data structure layer. We are not aiming at pinpointing elements in an ignored surrounding structure but in a detailed data structure de- and recomposition.

  Abstraction of control parallelism can be gained by specifying parallel control schemes as higher-order functions. Both help to build a collection of predefined abstract parallel schemes (with clear interfaces, demand behaviours, and parallel efficiencies) which can be applied in a plug-and-play fashion during program development.

- *Program development methods*

  There exist not many approaches to program development in functional languages, and we have only been able to find a single approach[140] for the development of parallel functional languages. Therefore a standard method for organising the development of Eden programs needs to be outlined.

1.2 Contributions

The contributions of this thesis are:

- We will introduce a portable preprocessor to the Eden implementation. This preprocessor is based on Template Haskell and allows for general and domain-specific program analyses and transformations. As an application we will describe Eden preprocessing stages using the preprocessor.

- We will introduce a new structure-oriented approach to generic parallel programming for Eden, which is based on the preprocessor. Additionally we will define a set of sequential and parallel generic functions.

- We will discuss and summarise possibilities to control demand in Eden. We will show a way to uniformly express demand exertion and propagation.

- We will enhance data-parallel skeletons by using generic methods. Generic functions for data structure partitioning will be given.

- We will define control-parallel skeletons for dealing with irregular tasks contained in dynamically evolving task sets. We will also provide a skeleton for introducing short-cuts into the regular Eden process tree.

- Finally we summarise our results in an outline of a program development method for Eden.

1.3 Structure of this Dissertation

The goals and contributions of the previous section are reflected in the dissertation structure shown in Figure 1.1. After the introduction, we will describe fundamentals needed for our work in Chapter 2. Chapters 3, 4, and 5 deal with introducing our three tools for enhancing programming in Eden: meta-programming for static program modifications, an approach to generic programming for flexible function definition, and demand control techniques for steering parallelism. The following two Chapters 6 and 7 apply these techniques to the two parallelism models data and control parallelism. Finally, the techniques shown before are united in Chapter 8 to a program development guideline for Eden. Chapter 9 discusses related work and Chapter 10 concludes.

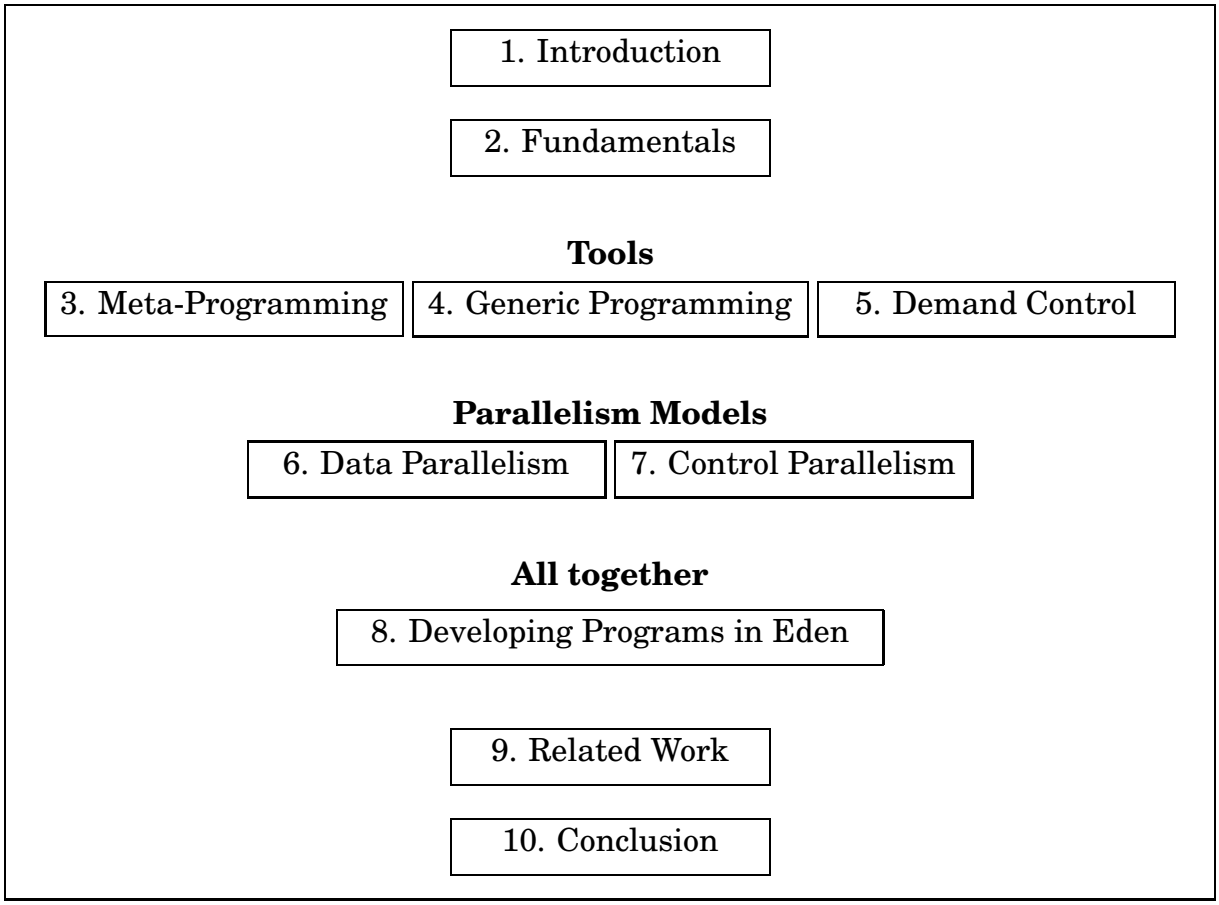


Figure 1.1: Structural Chapter Overview

2. Fundamentals

"A renaissance of parallel computing research is on the horizon since:

1. *chip multi-cores are the wave of the future for all major hardware vendors,*
2. *since 2003 clock frequency of CPUs is hardly advancing any more,*
3. *the 6-decade quest for an easy-to-program parallel architecture paradigm is yet to provide a competitive alternative to the serial paradigm."*

CFP of 18th ACM Symposium on Parallelism in Algorithms and Architectures

2.1 Motivation

Within this chapter, we provide a short panoramic view across the various types of parallel computers, the different non-functional approaches of programming them, and finally a survey of functional parallel programming approaches. After a short categorisation we describe *Eden* and show why Eden is a promising approach.

> Section 2.2 describes parallel computer architectures, while Section 2.3 and Section 2.4 describe first traditional non-functional and then functional approaches to programming these. Section 2.5 closes by describing the parallel functional language Eden together with a survey of research conducted for Eden.
>
> Parts of the chapter are based on two fundamental Eden articles [32, 143] .

2.2 Parallel Computers

Many important applications in science and technology (like simulations) need far greater computing power than common computers based on a single processor unit can deliver nowadays. Because algorithms often contain parts which can be computed independently from other parts, they lend themselves to simultaneous evaluation on different computers. This transition from *sequential* to *parallel* processing promises huge gains in program execution performance. However, for these gains one has to accept much more complex programs: Program sequences running in parallel cooperate regarding solving the problem, but compete regarding

the access to limited machine resources like processor time, memory, and periphery. Furthermore one has not only to ensure correctness of the sequential program parts, but now also correctness of their parallel interaction. The programmer has to deal with the efficient partitioning of the algorithm into mostly disjoint parts and the coordination of cooperation. Both depend heavily on the architecture of the parallel computer, on which the resulting program will be executed.

Before looking at programming, we want to have a short look at the technical side of parallelism, as advances in technology preceded and attracted advances in parallel programming research. Parallelism has for a very long time been an important part of a computing system's power, but has usually been hidden from the programmer. The older pipeline parallelism has been exploited by optimising compilers which reordered instructions to feed the pipeline in an optimal way; all this happened without the programmer's knowledge. Nowadays, however, processors are hitting a performance barrier which seemingly cannot that easily be overcome: Clock frequencies are stagnating, pipeline parallelism is limited, caches are getting larger and larger demanding bigger chip dies, and it seems that parallelism can no longer be hidden from the programmer if further performance gains have to be achieved. It seems no longer to be sufficient to speed up computations by exploiting the independencies on a small scale (like instructions): Parallelism has to be found at a larger scale which means bothering the programmer. Therefore, parallelism is nowadays not a side matter anymore.

To set the scene for our thesis, we will give a short classification of parallel architectures. The approaches to exploiting parallelism span many levels which are extremely different; a traditional classification by Flynn [62, 24] differs between the relation of instructions and data at execution time and defines four classes:

1) SISD (single instruction, single data).

This class comprises the classical PC, workstation, or single node of a parallel computer where a single instruction stream is applied to a single data stream. As there is no interprocessor parallelism, we will now shortly sketch parallel processor-internal techniques[87] :

- In a RISC architecture, *pipelining* is an essential feature by which the execution of a set of instructions can be accelerated. Each instruction is split into a set of execution phases and fed into a pipeline of execution units. As soon as the first phase of the first instruction has finished, the execution of the first phase of the second instruction can begin. This parallelism is limited, as instruction dependencies often prevent the pipeline from being completely filled.

- *Instruction-level parallelism* exploits independencies between instructions. Firstly, this can be done dynamically by a *superscalar* processor which has multiple execution units. Independent instructions are then scheduled at runtime to different units. Secondly, a program can also be statically analysed for such independencies. An optimising compiler then builds *very long instruction words (VLIW)* each containing several instructions which have been recognised as being independent. These are fed into a processor which

also has multiple execution units but no scheduling decision mechanism; it just executes the statically determined VLIWs.

- *Hyperthreading* is multithreading implemented in hardware. Pipeline stages and registers are duplicated to support multiple parallel threads.

Additionally, there are two new parallel processor trends which are processor-internal but could be put into other groups: *Graphical processing units (GPUs)* exploit massive data-parallelism contained in graphical representations and belong to the SIMD group. Also, the recent *multi-core* processors, which are mainstream technology by now, simply contain multiple single-processor cores acting on a shared memory. These are essentially multiprocessors of the MIMD group.

2) SIMD (single instruction, multiple data).

Within SIMD computers, a single instruction stream is applied to multiple data streams. Therefore one gets synchronous data-parallelism. This kind of parallelism is fine-grained and also called tightly coupled as after each instruction all nodes are synchronised. Communication happens via a collective data exchange. Vector and array processors are typical representatives of that class.

3) MISD (multiple instruction, single data).

If multiple instruction streams are applied to a single data stream, a pipeline results in which data traverses parallel nodes which each execute a stream of instructions on each data element. In contrast to the other three classes, there are almost no parallel architectures belonging to this group.

4) MIMD (multiple instruction, multiple data).

MIMD computers consist of nodes which each apply an instruction stream to a data stream. These computers are also called loosely coupled or asynchronous as no direct synchronisation is preassumed. If synchronisation becomes necessary, some effort is needed to do so; therefore, synchronisation is avoided by using more coarse-grained parallelism. In general one differs between two kinds of MIMD computers depending on the distribution of memory (also shown in Figure 2.1):

- *Multiprocessors* consist of a set of nodes which are connected via a network and share a common memory. Communication and synchronisation happen via reading and writing of shared memory locations. A disadvantage is the network, which presents a bottleneck for memory accesses causing memory contention. As each node usually manages an own cache, special cache consistency algorithms have to applied to avoid memory misinterpretations. In total, shared memory eases communication but aggravates network contention.

- *Multicomputers* consist also of a set of nodes connected over a network. In contrast to multiprocessors, each node is equipped with its own memory; there is no shared memory. Communication is therefore not implicit via shared memory but explicit via send and receive operations (message-passing). The distributed memory makes synchronisation (as every communication) more

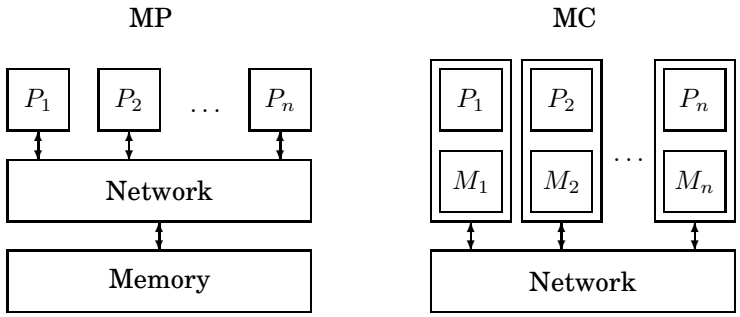


Figure 2.1: MIMD: multiprocessors and multicomputers

expensive. Therefore *data locality*, which means that the computations of a node can mostly be done with local data, is of fundamental importance in such systems. In total, multicomputers are a straight-forward model of parallel computing: Off-the-shelve nodes can be connected via inexpensive networks. Communication via message-passing is possible via standardised libraries[160, 76] and many problems can be solved by modelling them as a set of communicating programs.

In our thesis we will assume the presence of a multicomputer. These occur most often as *clusters* of conventional uniprocessor nodes. We will not aim at other architectures, like the recent *Grid*[63, 49] architecture: A grid can be seen as a cluster of clusters or also as a multicomputer whose nodes are parallel computers. Grids and cluster may be homogeneous or heterogeneous, which means that its nodes have either uniform speed, memory, and network connection or not. In our thesis we will not assume homogeneity.

Given these architectures, parallelism can appear in different degrees of granularity and abstraction[24] (see Figure 2.2). On the program level, multiple programs can run interleaved or even truly parallel. On the function level, processes or threads are used to model remote function invocation. Expressions can be executed in parallel on an instruction level. Finally, an instruction can often trivially be parallelised by executing the bitwise operations in multiple chip units.

Level	Parallelism unit	Example
program	job, task	operating systems, grid
function, procedure	process, thread	MIMD
expression	instruction	SIMD
bit	instruction internal	processor-internal

Figure 2.2: Levels of parallelism

2.3 Approaches to Parallel Programming

In this section we will shortly outline traditional parallel programming languages. The next section will then deal with parallel functional approaches.

There are numerous and very different approaches to parallel programming; almost equally numerous are categorisations of these. We will now give another classification and point out some approaches. To set the scene, we will first clarify three terms whose meanings are often intermixed. Each of them defines a different aim concerning coexisting activities:

- *Concurrent programming*[7] deals with the safe execution of tasks on a single node. These tasks can either be running, runnable, or blocked and compete for entering the running state. A scheduler distributes running time in the form of time slices (preemptive scheduling), or each task decides on itself to hand over control to another task (cooperative scheduling). This distribution should be fair, such that starvation cannot happen and progress is guaranteed for each task. The interleaved execution of all tasks raises the impression of true simultaneity if the time slices are small enough. The tasks have to be restricted concerning the access to the memory shared by them, such that no conflicts arise. This mutual exclusion (as well as synchronisation) is usually achieved via semaphores or monitors, which depend on uninterruptible (also called atomic) operations. A typical example for a concurrent program is an operating system managing different user tasks. In total, concurrency is about safe coexistence, fault avoidance, long runtimes; its tasks are independent but maybe collaborating[80, 13] and often pursue no shared aim.

- In contrast, *parallel programming*[175] is based on many (usually homogeneous) nodes, each of which concurrently runs a set of tasks. Tasks on different nodes can then be active at the same, which is the aim of parallelism: to speed up calculations by distributing them over multiple nodes and letting their runtimes overlap. Parallelism inherits much of the problems of concurrency mentioned above. In total, parallelism is about speed, and small to medium runtimes; its tasks are interdependent and collaborate to solve a shared aim.

- *Distributed programming*[146, 169] on the other hand has the same execution environment, but is designed for coping with heterogeneity and change. This means, that different nodes connected with different networks (and latencies) cooperate in the sense of concurrency. In total, distribution is about long runtimes, fault recovery, and dynamic node changes. An example of a distributed program is a multiuser video conference system run in the Internet within which users can join or leave the conference.

Given one of these aims, there are three design approaches (as argued in Chapter 29.3 of Zomaya[217]) for a new language:

- The one which bothers the programmer the least is to *enhance an existing sequential language compiler to detect parallelism*. Program analysis is then

completely responsible for discovering parallelism in a sequential program and executing it efficiently.

- One can also *introduce explicit parallel constructs into an existing sequential language*. This has the advantage of building upon a sequential base language compiler.
- Instead of integrating parallel constructs into a language, one can also *attach a parallel coordination language to an existing sequential language*. The sequential language will be used for defining the computations whose execution on a parallel computer is then managed by separately defined coordination instructions.
- The most straight-forward method, however, is to *invent a completely new language*. This has the advantage of getting a homogeneous language, but the disadvantage of having to implement a whole new compiler.

Now we can give a set of parallelism views into which parallel languages can be sorted. The following presentation is a blend of the many parallel language surveys[194, 64, 217] that have been already published:

Shared Memory. These approaches are dominated by the idea of a globally accessible shared storage, via which all tasks communicate and synchronise. This is typical for operating systems (like Unix[191]) with processes, threads, locks, and barriers. OpenMP[20] is a typical representative.

Message-Passing. These models see the world as a set of tasks communicating synchronously or asynchronously via send and receive operations. Such models are typically implemented as a library and enrich an existing sequential language. Representatives are the PVM (Parallel Virtual Machine)[160] and MPI (Message Passing Interface)[76] libraries. The languages Occam[134] , Fortran M[64] , and Erlang[11] also belong to this group.

Data-Parallelism. The data-parallel model is driven by data structures, especially arrays. Processes and communication are not immediately present, but parallel matrix operators as well as complex partitioning operators. SIMD machines are typically used to execute program written in such languages. To these belong: Fortran 90 and its successor High-Performance Fortran[64] , NESL[19] , and C-Star (Thinking Machines Corporation).

Communication Space. These models are similar to the ones using shared memory, but instead of a global storage a global communication space is assumed. The most famous approach is Linda[37] , which administrates a tuple space, in which active process tuples and passive data tuples meet.

Skeletal. Skeletal programming models attach high-level programming constructs to a sequential language. These skeletons[45, 55, 56] can either be general or specialised to certain parallel architectures. Skil[22] , for example, attaches skeletons to an imperative language with functional features like higher-order functions and a polymorphic type system. P3L[52] is a similar approach based on C.

Object-Oriented. These models use objects as parallel units. Typical representatives are the Actors[2] model and Compositional C++[64] .

Logic. There are also approaches which parallelise the evaluation within logic and constraint languages. A member of the first group is Strand[65] .

Functional. The group of parallel functional languages will be discussed in detail in the next section.

Such an enumeration can hardly be complete. See Figure 2 of Skillicorn's survey[195] for another classification and enumeration of parallel approaches.

2.4 Approaches to Parallel Functional Programming

Approaches to parallel functional programming can be categorised in many ways, because the combination of functional programming and parallelism possesses so many facets which can vary. If one (or sometimes two) facet is considered most important, the ordering of approaches with respect to that facet yields a new classification. Common ones are:

- Strictness
- Purity
- Level of control (automatically or by hand)
  - Task definition
  - Data decomposition
  - Physical mapping
  - Communication
  - Synchronisation

In our presentation, we will restrict ourselves to non-strict and purely functional approaches. For others, see the presentation in Figures 1.5 and 1.6 of Hammond et al. [80] and the discussion in Section 6.2 of Loogen et al. [143] . Therefore, we will classify approaches after the level of parallel control, as proposed in Figure 2 of Skillicorn's survey[195] . We will classify only functional languages, although there are also many non-functional representatives for each group:

Nothing explicit, parallelism implicit. Models in which no parallel construct whatsoever is used and where parallel execution is not even guaranteed.

Representatives: Evaluation transformers[142] , PMLS[78]

Parallelism explicit, decomposition implicit. Models in which potential parallelism is marked, but not necessarily fully executed. The runtime system selects the right amount of parallelism and cares for mapping, communication, and synchronisation.

Representatives: GpH[83, 205] , Concurrent Clean[156]

Decomposition explicit, mapping implicit. Models in which parallelism and decomposition are explicit. Mapping, communication, and synchronisation are implicit. The work is partitioned explicitly into pieces, while execution details are automated.

Representatives: Eden[143]

Mapping explicit, communication implicit. Like the former, but with explicit mapping. Defining the physical mapping in the program commits it to a special parallel architecture and damages portability.

Communication explicit, synchronisation implicit. In these models, only synchronisation is left implicit. These differ not much from the last category:

Everything explicit. Everything is handled explicitly by the programmer.

Representatives: Haskell with MPI[33]

We will now turn to describing the language Eden, which we have sorted into the third group.

2.5 The Parallel Functional Language Eden

This section describes the parallel functional language *Eden* and follows the presentation in our diploma thesis[171] . Eden's characteristics are:

- **Parallel extension of Haskell.** Eden is a clear parallel extension of the standardised lazy functional language *Haskell* [164] . It introduces parallelism via two constructs which are intuitive in use due to their duality to λ-abstraction and λ-application. The language inherits Haskell's laziness with all its advantages like non-strict functions and infinite and circular data structures. Being an obstacle to parallelism, laziness also causes drawbacks which are dealt with via explicit demand control.

- **Semi-explicit.** Eden belongs to the semi-explicit approaches, as it delivers enough control for writing efficient programs while not burdening the programmer with every detail of parallel coordination. Process construction is explicit, and allows for the construction of arbitrary process systems if Eden's dynamic channels are used for introducing cross connections in the process tree. Communication on the other hand is implicit.

- **Semantics.** Eden has a clearly defined operational semantics together with an abstract machine execution model.

- **Implementation.** Eden has been implemented as an extension of the standard *Glasgow Haskell Compiler GHC*[165] .

Different topics concerning Eden have been covered by a wide range of research in the past:

- The language has been developed in various articles[25, 32, 28, 26, 143] and compared to other approaches in another one[137] .
- On the programming side, Eden has been successfully used for programming with skeletons[69, 31, 119, 141, 149] . Also, dynamic channels have been used for creating complex communication structures[16] and Template Haskell for adapting to a parallel execution environment[79] .
- The implementation of Eden has been treated both theoretically [30] and practically[27, 171, 29, 33, 15] . A very recent development is the reimplementation[14] of the GHC runtime system in Haskell to form a generic parallel runtime system which is capable of supporting also other parallel Haskell dialects beneath Eden.
- A tool for tracing the behaviour of a running Eden program has been implemented in Java[72] and reimplemented in Haskell[198] with extended functionality.
- Formally Eden has been treated in terms of program analyses[120, 187, 204] and semantic definitions[91, 90] .

In our thesis, we will concentrate on the expressiveness of Eden. The next subsection will introduce Eden's features step after step.

2.5.1 Process Creation and Placement

Eden extends Haskell with means for relocating the evaluation of a function application to another network node, enabling the evaluation of several expressions in parallel.

2.1 Definition (Process Abstraction and Application)
A function embedded in a process abstraction *by applying*

```
process :: (Trans a, Trans b) =>
           (a -> b) -> Process a b
```

can be run in parallel to the continuing evaluation of its parent expression on another processor by applying its arguments to a special application operator

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

A process abstraction therefore serves as a process template which can be instantiated in many ways.

The type variables imply that all values for which corresponding low-level sending and receiving functions exist (ensured by the `Trans` context) can be communicated. Given a process abstraction `Process a b` the inputs `a` and `b` are often tuples. The elements of the first one are called *inports* and of the second one *outports*. If a process is created out of an abstraction, a thread for evaluating every inport is

created on the parallel node of the creator. Correspondingly, a thread for each outport is started on the node where the process is placed. A bit deeper within Eden's implementation the `(#)` operator is directly implemented via

```
createProcess :: (Trans a, Trans b) =>
                 Process a b -> a -> Lift b
data Lift a = Lift a
```

Wrapping the result in a data constructor is necessary to keep local evaluation from stopping because of waiting for the process to deliver a result. By lifting the result its head-normal form can immediately be obtained. Thus subsequently created processes will run in parallel to the first one which would not be the case without lifting[118] . We will further discuss that matter in Chapter 5.

Being embedded in expressions which are tree-shaped, process creations also form a tree-shaped process and communication structure. As every process can create many child processes, the natural Eden process topology is a rose tree. Due to its non-strictness, cycles and mutual dependencies can be introduced into the tree. Note that Eden only defines a *logical* process structure and does not assume a certain kind of parallel architecture. The efficient physical mapping of processes to parallel nodes lies within the responsibility of the Eden implementation, making Eden architecture-independent. The usual mapping strategies are random and round-robin distribution.

For the sake of efficiency, however, it has proven nevertheless useful to be able to exert some control concerning process placement. The `createProcess` function above is itself implemented via `createProcessAt`, which has an additional parameter identifying the number of the node on which the new process has to be created. `createProcess` itself signals via a negative value, that node selection has to be done by the runtime system:

```
createProcess = createProcessAt (-1)
createProcessAt :: (Trans a, Trans b) =>
                   Int -> Process a b -> a -> Lift b
```

Explicit placement is useful, if a specific physical mapping for a known parallel architecture has to be enforced.

2.5.2 Communication and Storage

In Eden, every exchange of data between running processes happens via communication. Every parallel node administrates its own local heap. There is no globally shared heap, and therefore no global references which would imply the implementation of a distributed garbage-collection. Communication then causes expressions from one local heap to be sent and inserted into another local heap, where it replaces a heap construct which represents an absent value onto which demanding threads block until it is inserted. This means, that sending is non-blocking and receiving is blocking; therefore, processes can only synchronise by exchanging data via communication. Communication connections are established when a process is created. Via a connection three kinds of values can be communicated:

- A *single finite value*
- A potentially infinite *stream of finite elements*
- A potentially infinite *stream of potentially infinite elements* (also called *channel structures*[28] and currently not implemented)

The second alternative transmits values elementwise and is often used to build a system of interdepending streams. For managing these stream systems, it is often necessary to merge two streams into a single one. For doing so, Eden defines a merging process

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

Although being initially defined as a separate process, `merge` has today become a function

```
merge :: Trans a => [[a]] -> [a]
```

This eliminates the cost of setting up an additional process and directly merges the streams at the receiver process. However, when using `merge` one has to be careful since it introduces non-determinism into the language. But even then one can keep up functional purity in most parts of a program; by carefully sorting the merged streams (see Section 7.2), element order can be regained.

We have mentioned above, that in Eden the normal process system is tree-shaped. As this structure contains long communication ways (without the bypassing mechanism[120]), *dynamically created channels* were introduced for establishing additional interconnections between processes. An Eden process can create a new dynamic channel by using the `new` function:

```
new :: Trans a => (ChanName a -> a -> b) -> b
```

A call

```
let (ch,in) = new (\ch val -> (ch,val)) in ...
```

then provides a channel name `ch` as well as an input value `in`. The channel name can be sent via regular communication to find its communication partner; if another process receives the channel name, it can pass it on or use it himself. The values fed into the channel will then appear under the name `in` at the receiver. A dynamic channel is used via the `parfill` function. Its name indicates, that a channel is filled in parallel to the evaluation of a local expression:

```
parfill :: Trans a => ChanName a -> a -> b -> b
```

It creates a thread (as a side effect) which feeds its second argument into the channel represented by the first argument; concurrently, the original evaluation thread continues with the third argument. To continue our example from above:

```
parfill ch some_values continuation
```

will send `some_values` into the channel concurrently to evaluating `continuation`. Using these dynamic channels, powerful process structures can be set up (see Berthold and Loogen[16] and Section 7.4). One has, however, to take into account that some build-up time is necessary to set up the regular process tree and propagate channel names through it to establish the additional connections.

Processes are terminated if either every outport has produced its complete output or if a garbage-collection at the receiver node determines that further results of the process are not needed. In either case, all inport threads and all outport threads are stopped. This again may cause other processes to terminate, as their outputs are now as well unneeded. Therefore termination may propagate through the process system.

2.5.3 Evaluation Behaviour and Laziness

As mentioned above, Eden integrates parallelism and laziness. While laziness aims at postponing evaluations as long as possible, parallelism relies on early process creation to gain overlapping process lifetimes. We will discuss this matter in Chapter 5 thoroughly and will show how to control laziness such that timely process creation can be achieved. In short: One has to explicitly steer demand to a certain degree to gain early process creation. Even more, strict evaluation of outports and strict sending of values without caring for their neededness, is advantageous in Eden.

Additionally one has to be aware, that by introducing the strict evaluation of outports a chain of process creations may be triggered. As we will also show in Section 5.4, demanding an outport (out of a tuple of outports) causes a variety of demand on other expressions: Demand will not only be exerted to all inports, but also to all remaining outports. All this contains the danger of runaway parallelism, which is a flood of speculative processes.

3. Meta-Programming for Eden

"One of the most under-used programming techniques is writing programs that generate programs or program parts."

Jonathan Bartlett, 'The Art of Metaprogramming'

3.1 Motivation

Suppose you have designed a new, fancy domain-specific extension of Haskell[164] , possibly an extension for providing easy data base access, for interfacing to foreign languages, for providing faster computations by exploiting parallelism, or for mobile computing. How do you implement your extension? Developing a whole new compiler would mean reinventing the wheel, while extending an existing Haskell compiler most often produces a deep entanglement of compiler interiors with domain-specific implementation parts. That would result in reduced portability and maintainability, which is especially problematic if the base compiler is subject to frequent version changes which usually have to be reproduced:

$$\begin{array}{ccccc} \text{host language} & & \text{host implementation} & & \text{v1.0} \rightarrow \text{v1.1} \rightarrow \text{v2.0} \rightarrow \ldots \\ \downarrow & \Rightarrow & \downarrow & & \\ \text{extension} & & \text{extension implementation} & & \text{v1.0} \rightarrow \ldots\ldots \end{array}$$

Having all your implementation extensions contained in a traditional library seems like a nice idea. But a mere collection of subroutines would not be powerful enough, since such extensions usually also need changes to the runtime system and new optimisation passes run by the compiler. Following the idea of *active libraries*[50] one solution to this could be to use meta-programming tools to separate the domain-specific implementation from the compiler while expressing it as a library. Steele [196] considers this an "interesting language design strategy":

> "Wherever possible, consider whether a proposed language feature can be provided by a library rather than having it wired into the compiler."

This envisions having a compiler for a base language and extensions of that language implemented as a set of attached libraries.

Eden[143] is such a domain-specific extension of Haskell which introduces constructs for parallel programming. Its first implementation consisted of a largely

modified *GHC*[165] which was very hard to maintain over GHC version changes. To avoid these problems parts of the Eden-specific implementation have been pulled out[15, 14] of the GHC. Among others one thing that still remains to be separated from the GHC is the preprocessing machinery for analysing and transforming Eden code which still resides amidst GHC's original code. This can also often be the case for other domain-specific extensions.

Besides writing a library one could also suggest building a standalone tool for preprocessing the sourcecode which can then be fed into the compiler manually. Disadvantages include: the introduction of an extra tool which needs additional care, manual and possibly erroneous operation by the user, the inability to use compiler facilities (functionality needs to be rewritten) and compile-time information and therefore probably a large reimplementation of compiler machinery. Much more promising instead is a preprocessor library which remains separate but can be glued on top of the compiler. The same has been done for the foreign function interface *Greencard* by making the transition from an external tool[157] to an integrated (active) library[177] .

We will use compile-time meta-programming facilities provided by *Template Haskell*[189] to implement a series of pre-GHC preprocessing passes on domain-specific source code. The preprocessor code will be automatically inserted into the source program and run prior to the compilation of the actual program, meaning that an enriched program is created which preprocesses itself. We will show how the new implementation (forming an active library) can be separated from the GHC implementation, while only a small hook for the invocation of the preprocessor needs to remain. The scheme shown is generally applicable and in no way tied to the Eden implementation. Advantages include a simplified addition of preprocessing passes, shorter and more concise pass code, and enhanced maintainability. The scheme will be used to rewrite and shorten existing passes and in the next Chapter to express a kind of generic parallel programming in Eden.

We will also use the preprocessor to implement program transformation passes. Especially those transformations within which a program inspects itself (also called *reflection*), are interesting and will be examined.

In the next Section 3.2 we will introduce Template Haskell as our tool for meta-programming. The following Section 3.3 will then describe in detail an Eden preprocessor implemented in Template Haskell. Subsections cover the main idea, technical implementation issues, a state transformer monad needed for the implementation, global actions during preprocessing, and the preprocessor's main loop. Afterwards Section 3.4 describes a basic meta-programming application, namely instance derivation. Finally Section 3.5 describes other useful applications of meta-programming to Eden: program transformations and self-configuring skeletons.

Parts of the chapter are based on our GPCE paper[172] .

3.2 Meta-Programming with Template Haskell

Active libraries can be built using *Template Haskell*[189] , which is a typesafe compile-time meta-programming extension of Haskell built into the GHC. Essentially Template Haskell permits the definition of functions which can take Haskell code as arguments, yield Haskell code as their result and are typechecked and executed during compilation of the actual Haskell program. Therefore it introduces a second layer of execution by allowing to label Haskell expressions as "to be executed by interpretation at compile-time". This means that one can insert Haskell code that is meant to be run at compile-time into regular Haskell code:

$$\ldots \text{Runtime Haskell} \ldots \underbrace{\texttt{\$(}\ldots \text{Compile-time Haskell} \ \ldots\texttt{)}}_{Splice} \ldots \text{Runtime Haskell} \ldots$$

The result of this *splice* expression then has to be Haskell code described in an abstract syntax which will replace the splice and be embedded as runtime Haskell into the surrounding code. This corresponds to classical macro expansion, except that the newly generated Haskell code will also be successively typechecked. This is possible because splices are expanded by the GHCi interpreter during the typechecking phase.

A set of data structures (see Figures 3.1 and 3.2) forming an abstract syntax of Haskell is defined to be able to handle Haskell code as a value. There are types for patterns (`Pat`), expressions (`Exp`), declarations (`Dec`), types (`Type`) and so on. A function declaration consists of a name and a set of clauses. Each clause (which is a function alternative) contains of a list of single patterns, a body, and a list of declarations which represents the `where` block. A body can be guarded or unguarded and finally contains the body expression.

Datatype declarations via `data` are described by `DataD` and contain a type context (which is a list of `Type`), its name, the type variables, the constructor alternatives defining the datatype, and finally the derived type classes. An instance is declared similarly: a context is followed by a type (containing the type class name and the instance type) and a set of declarations which define the instance's functionality.

Depending on its position the code inside a splice has type `Q Exp` (part of a bigger expression) or `Q [Dec]` (top-level declarations). The `Q` monad is introduced among other things for encapsulating the generation of fresh names. The code within a splice has to be wrapped by the `Q` monad.

A simple case expression for instance could then be described like this:

```
(CaseE (AppE (VarE "f") (VarE "x")) [...]) :: Exp

[| case (f x) of ... |] :: Q Exp
```

The first row shows pure abstract syntax while the second row uses the *quasi-quotation* operator `[|.|]` for automatic transformation of user-legible code into abstract syntax. Quasi-quotation occurs inside a splice. Both versions can be used although the first one is more expressive than the second. The same can be done for declarations using `[d|.|]`. A binary tree declaration

```
data Lit = CharL Char | IntL Int | ...

data Pat = LitP   Lit                           -- literal
         | VarP   String                        -- variable
         | TupP   [Pat]                         -- tuples
         | ConP   String [Pat]                  -- constructor patterns
         | TildeP Pat                           -- lazy pattern
         | AsP    String Pat                    -- as pattern (@)
         | WildP                                -- wildcard (_)
         | RecP   String [FieldPat]             -- record
         | ListP  [ Pat ]                       -- list

data Exp
  = VarE      String                            -- variable
  | ConE      String                            -- constructor
  | LitE      Lit                               -- literal
  | AppE      Exp Exp                           -- function application
  | InfixE    (Maybe Exp) Exp (Maybe Exp)       -- infix / partial app
  | LamE      [Pat] Exp                         -- lambda abstraction
  | TupE      [Exp]                             -- tuples
  | CondE     Exp Exp Exp                       -- if then else
  | LetE      [Dec] Exp                         -- let decs in exp
  | CaseE     Exp [Match]                       -- case e of alts
  | DoE       [Stmt]                            -- monadic do
  | CompE     [Stmt]                            -- list comprehension
  | ArithSeqE Range                             -- arithmetic sequence
  | ListE     [Exp]                             -- lists
  | SigE      Exp Type                          -- typed expression
  | RecConE   String [FieldExp]                 -- record
  | RecUpdE   Exp [FieldExp]                    -- record update

data Match = Match Pat Body [Dec]

data Dec
  = FunD String [Clause]                        -- function
  | ValD Pat Body [Dec]                         -- p = b where decs
  | DataD Cxt String [String] [Con] [String]    -- data
  | NewtypeD Cxt String [String] Con [String]   -- newtype
  | TySynD String [String] Type                 -- type synonym
  | ClassD Cxt String [String] [Dec]            -- class
  | InstanceD Cxt Type [Dec]                    -- instance
  | SigD String Type                            -- type signature
  | ForeignD Foreign                            -- imported function
```

Figure 3.1: Haskell program representation in Template Haskell, (1/2)

```
data Clause = Clause   [Pat] Body [Dec]
data Body   = GuardedB [(Exp,Exp)]            -- guarded function body
            | NormalB  Exp                    -- normal function body
data Con    = NormalC String [StrictType]           -- normal constr.
            | RecC    String [VarStrictType]        -- recursive con.
            | InfixC  StrictType String StrictType  -- infix con.
data Module = Module [ Dec ]
data Type   = ForallT [String] Cxt Type -- forall vars. ctxt => type
            | VarT String               -- type var
            | ConT String               -- type constructor
            | TupleT Int                -- tuple types
            | ArrowT                    -- ->
            | ListT                     -- list type
            | AppT Type Type            -- type application
```

Figure 3.2: Haskell program representation in Template Haskell, (2/2)

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

could then look like this (again written in two ways):

```
[DataD [] "Tree" ["a"]
  [NormalC "Leaf" [(NotStrict,VarT "a")], NormalC "Node" ...] []
] :: [Dec]

[d| data Tree a = Leaf a | Node ... |] :: Q [Dec]
```

We will elucidate both splicing and quasi-quotation by a simple example:

---

1 Example (Selection from n-tuples)
Consider writing a function `select` which returns the *i*th value of an *n*-tuple such that `$(select 2 4) (a,b,c,d)` returns `b` (see Sheard and Peyton Jones[189]):

```
select :: Int -> Int -> Q Exp
select i n = [| \ x -> $(return (CaseE (VarE "x") [alt])) |]
  where alt  = Match pat rhs []
        vars = ["v"++(show j) | j <- [1..n]]
        pat  = TupP (map VarP vars)
        rhs  = NormalB (VarE (vars !! (i-1)))
```

A lambda abstraction is generated which contains a `case` for deconstructing the tuple into its components v_1 to v_n before v_i is selected and returned. Note the nesting of another splice into the quasi-quotation.

Quasi-quotation cannot only be used for the simple construction of code but also for its deconstruction as the contained code will just be translated into abstract syntax. Therefore we can write

```
do { abssyn_case <- [| case (f x) of ... |]; ... }
```

to extract the abstract syntax representation of the `case` expression which can then be used inside the `Q` monad and be spliced back in modified form. The same applies to declarations which can be decomposed by the `[d|.|]` operator.

In summary Template Haskell makes it possible to write Haskell programs which modify themselves at compile-time, which is exactly what we need.

3.3 Preprocessing Eden

Having introduced our meta-programming tool we will now turn to building the preprocessor. What we want to achieve is a preprocessor which takes a program, applies a series of preprocessing steps to it and places back the result. We want the preprocessor implementation to be separated from the base compiler implementation but at the same time have it glued closely enough to it to avoid getting an extra tool. We want the preprocessor to run on original code not yet altered by the compiler; this means, that no compiler internal transformation or simplification has been carried out yet.

3.3.1 The Preprocessor

We have seen that the quasi-quotation mechanism of Template Haskell can also be used to deconstruct declarations and expressions. Then why not let it embrace and decompose a whole program? This suggests the following basic scheme for preprocessing code with Template Haskell:

1. Textually embrace the given source code with quasi-quotes to get an abstract syntax representation of the source code.

2. Let predefined preprocessing machinery work on the extracted code given in abstract syntax (provided by the base compiler's parser).

3. Surround all this with a splice which delifts the modified source from abstract syntax back to regular Haskell code.

How can we operationally integrate that into a compiler? Simply by modifying the compiler to textually insert two small predefined code blocks into the source program before anything else happens. The source program will then carry around the preprocessing code for modifying itself through the compilation. The preprocessor will be triggered before other regular compilation stages are started.

In Figure 3.3 the transition of regular source code to an identical code with embedded preprocessing is shown. The two additional preprocessing code blocks contain `import` statements to import the Template Haskell module, a `Stager` module which contains the actual preprocessor functions, and a `Tools` module. User imports remain untouched. We assume that no module name clashes happen and

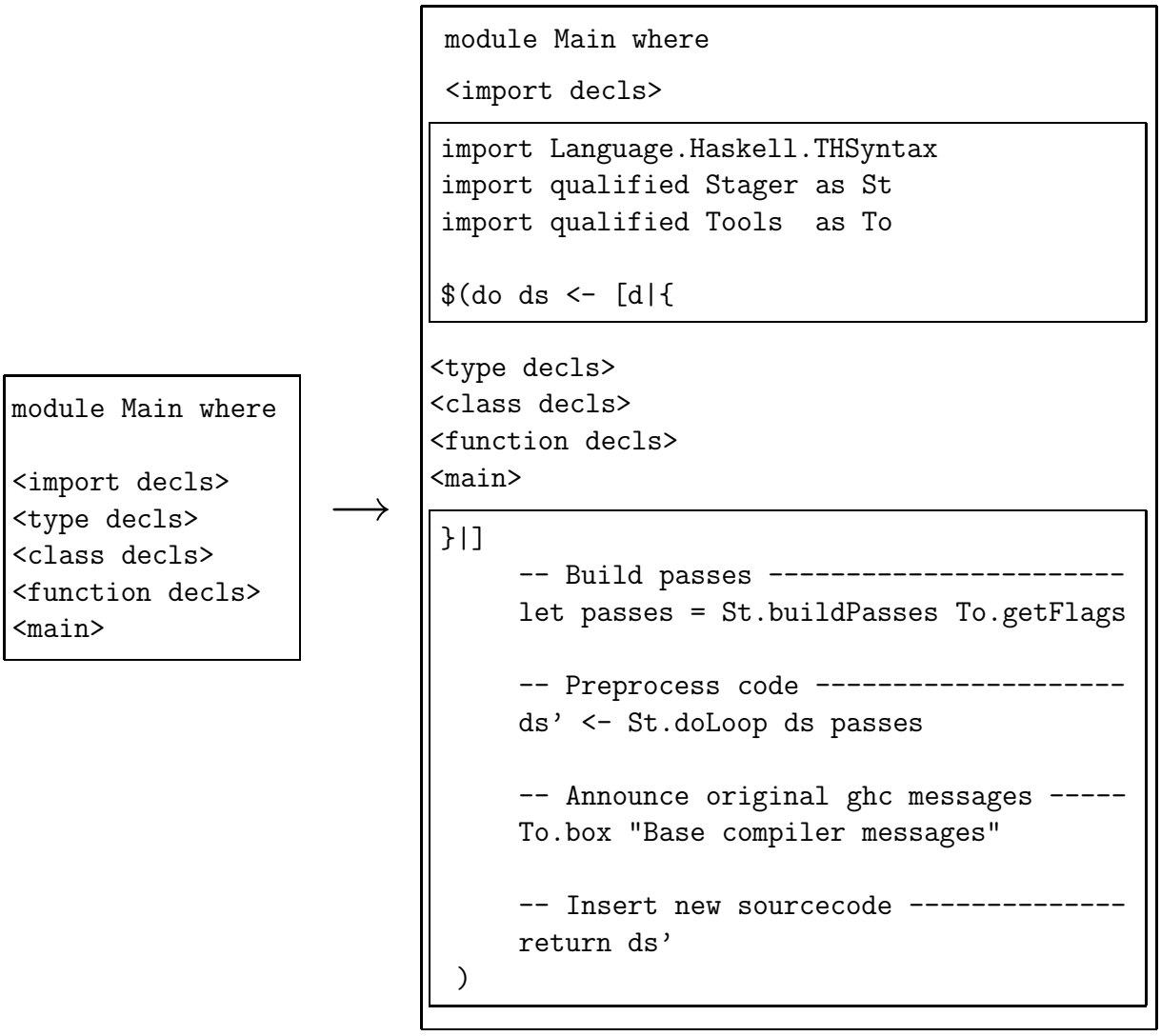


Figure 3.3: Transition from input source code to code with embedded preprocessor

therefore no renaming is necessary. The remaining code is contained inside quasi-quotation brackets. After extraction into an abstract syntax representation `ds` it has to be decided which preprocessing passes will be run on the code. External flags are read by `getFlags` and fed into `buildPasses` which will build a list of preprocessing passes. `doLoop` then runs these passes on `ds` producing the modified code `ds'`. After the loop has ended an announcement is printed that all the following messages belong to the underlying Haskell compiler. At last, the modified code `ds'` is returned and reinserted by the surrounding splice.

This solution permits the smooth integration of the preprocessor into the compiler implementation with only slight modifications while achieving technically a complete separation. In the following subsection we will discuss implementation aspects as well as advantages and drawbacks of the approach.

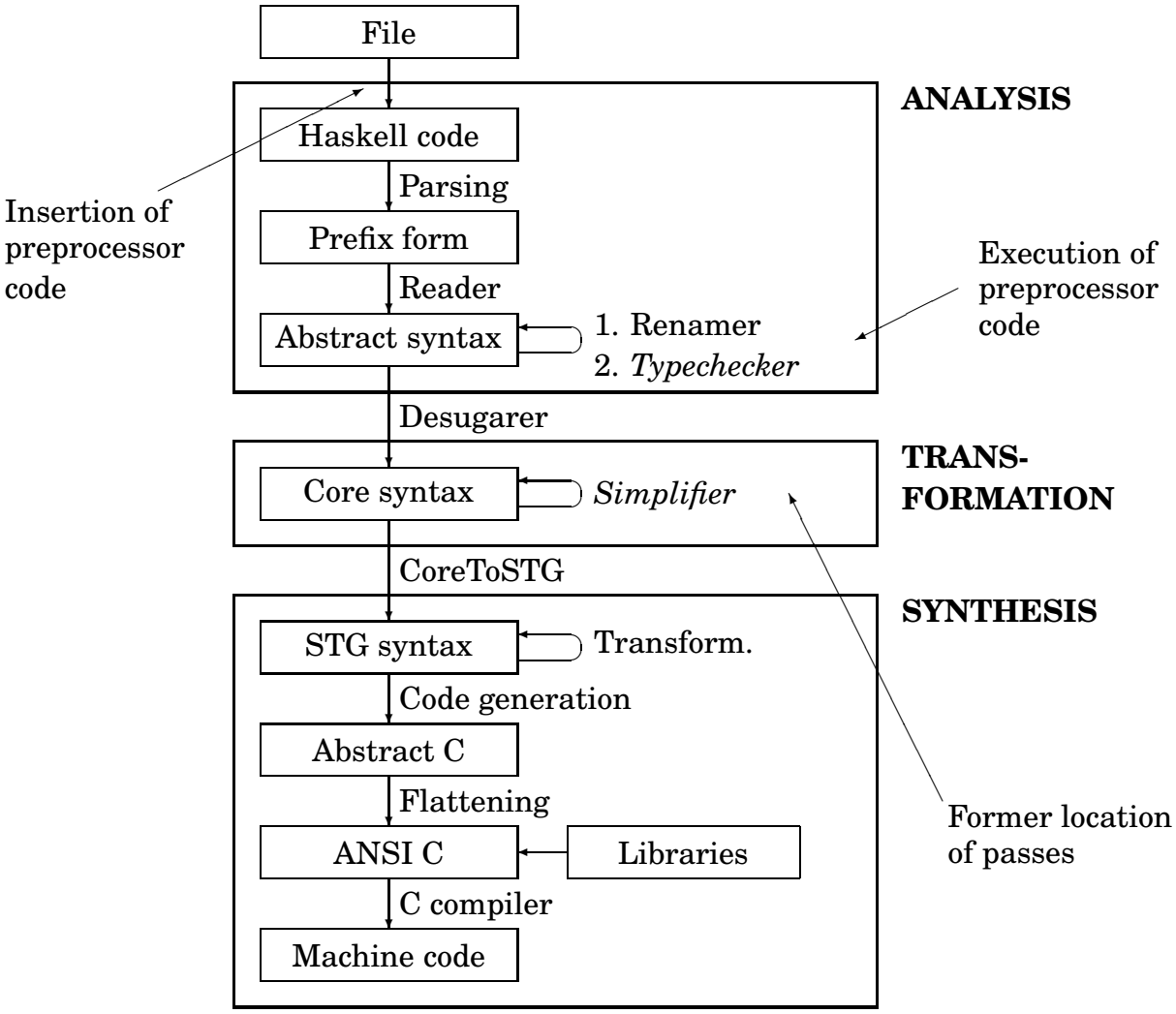


Figure 3.4: GHC workflow diagram with changes for preprocessor

3.3.2 Technical Issues

Figure 3.4 gives an overview of the workflow inside the Glasgow Haskell Compiler (GHC), which is the basis of the Eden compiler and into which we will therefore integrate the Eden preprocessor. The compiler stages can be roughly divided into the classical sections[3] analysis, transformation and synthesis. Until now additional analysis and transformation passes were usually placed in the transformation section and worked on the Core syntax representation, which is a subset of Haskell plus explicit type annotations and resembles the polymorphic lambda calculus. Due to complex functions and data structures for handling Core, inserting an additional pass is a complex task; furthermore, the extension would reside amidst the original compiler code. And with desugaring already having taken place it is additionally very hard to issue meaningful comments if necessary because the reference back to the original code is lost. A better place for domain-specific optimisation passes is the analysis section where the full source code is still available

and for example error messages of the preprocessor can contain much more accurate position descriptions. For domain-specific extensions it is especially important that their high-level constructs have not already been boiled down to less meaningful constructs. Our new preprocessor code will be inserted into the source program before any other GHC stage has started.

As a technical remark: This can be done by textually inserting the preprocessor calls into the program when it is read by the compiler. The driver, which steers the compilation process, works as a kind of pipeline which applies all necessary compilation steps to a program. There we can insert the preprocessor code at an early point in time.

Unfortunately there are also three drawbacks to the taken approach which partially are related to Template Haskell and may be removed in a future version:

1. Quasi-quotation cannot handle source code which completely relies on layout. Braces and semicolons have to be used sometimes.

2. Regenerating the modified code via splicing destroys the internal location tracking of GHC. Regular errors discovered by GHC will not contain a line number but only the message `<at compiler-generated code>` which makes it harder to track down errors.

3. Since the preprocessor acts outside the base compiler's passes, it has no access to informations (like types) gained by the base compiler. If these are required for an earlier pass, they have to be provided. On the other hand, one could also think of moving passes into the preprocessor proposed here.

At the bottom line there is a clear trade-off when using the preprocessor: On the one hand we are able to work on the richer full Haskell syntax and to formulate different passes, while on the other hand we have to face technical restrictions.

3.3.3 The State Transformer Monad

For most preprocessing passes an internal state is needed. The state can be used to collect information for passes which cannot work only on local information. To achieve an omnipresent state during a pass, the whole syntax tree traversal needed for a pass will be based on a predefined *state monad* `(ST s) s` (see the standard papers[213, 129, 130, 131, 113]) which is shown in detail in Figure 3.5. Each preprocessing pass will then traverse the syntax tree of the program to be preprocessed monadically. State modifications will then be strewn into the traversal; mostly the state will be passed on without being changed.

A *state transformer* `ST` maps a state `s` to a new state and yields an additional value of type `a`. The `MonadState` class captures those monads, which do not only provide the sequencing mechanism via `(>>=)` and `return`, but also functions for direct state transformation. `get` reads and returns the current state, while `put` overwrites the current state with a new state. `update` is similar to `get` and `put` but takes a state-transforming function, applies the function to the state, and returns the new state. The state transformer `ST` is both an instance of `Monad` and `MonadState`.

```
data ST s a = ST (s -> (a, s))

class (Monad m) => MonadState m s | m -> s where
   get    ::             m s                -- read state
   put    ::        s -> m ()               -- write state
   update :: (s -> s) -> m s                -- update state by function

   get = update id

instance Monad (ST s) where
  (ST m) >>= f = ST (\s -> let (v, s') = m s
                               ST m'   = f v
                           in m' s')
  return v     = ST (\s -> (v, s))

instance MonadState (ST s) s where
   put s    = ST (\_ -> ((), s  ))
   update f = ST (\s -> (s,  f s))
```

Figure 3.5: State monad definition

We will use this state transformer to model our preprocessing passes. Within a pass, the specific state of the pass can be changed anytime by using the functions of `MonadState`.

3.3.4 Stateful Monadic AST Traversal

We have seen how the preprocessor is attached to the source program and how it is run. But what should it do when it is run and how can that be specified?

Quasi-quotation delivers the program to be preprocessed as an abstract syntax tree (AST) of type `[Dec]`. Therefore we define a general type class `Traverser` (see Figure 3.6), which contains functions for the recursive traversal of that tree. For each part of the mutually-recursive abstract syntax (`Decs`, `Dec`, `Exp`, ...) shown in Figures 3.1 and 3.2 the class contains a corresponding transformation function (`tDecs`, `tDec`, `tExp`, ...). The `tMain` function is the central starting point which eventually calls `tDecs` and is therefore wrapped by the `Q` monad. By default all these functions are defined to return the unaltered syntax tree. As later each instance will represent a preprocessor pass, selected functions will be overloaded to examine or modify the tree if certain parts of special interest are encountered.

If the transformation functions were defined in a straightforward fashion

```
tExp (LetE ds e)  = do ds' <- mapM tDec ds
                       e'  <- tExp e
                       return (LetE ds' e')
```

one could not easily define small modifications, as for example the `Exp` datatype consists of many data constructors. If one would like to change only the behaviour

```
class (MonadState m s) => Traverser m s where

  -- Functions ------------------------------------------------------
  tName  :: m s -> String
  tMain  :: m s -> [Dec] -> Q ([Dec], [Pass])

  tDecs  :: [Dec]  -> m [Dec]
  tDecs' :: [Dec]  -> m [Dec]

  tDec   :: Dec    -> m Dec
  tDec'  :: Dec    -> m Dec

  tBody  :: Body   -> m Body
  tBody' :: Body   -> m Body

  tExp   :: Exp    -> m Exp
  tExp'  :: Exp    -> m Exp
  ...

  -- Defaults -------------------------------------------------------
  tName _            = "Identity"
  ...

  -- Decs defaults --------------------------------------------------
  ...

  -- Dec defaults ---------------------------------------------------
  ...

  -- Exp defaults ---------------------------------------------------
  tExp               = tExp'

  ...
  tExp' (VarE vname) = VarE vname                    -- variable
  tExp' (AppE e1 e2) = do e1' <- tExp e1             -- application
                          e2' <- tExp e2
                          return (AppE e1' e2')
  tExp' (LetE ds e)  = do ds' <- mapM tDec ds        -- let
                          e'  <- tExp e
                          return (LetE ds' e')
  ...
```

Figure 3.6: Monadic traversal class `Traverser` (excerpt)

of tExp for LetE, all other definitions, although being unaltered, would have to be rewritten. To avoid these extensive redefinitions when overloading only one alternative of a transformation function, each traversal function is split into two layers. For example, the function tExp is accompanied by a second function tExp'. By default tExp' is defined to continue the AST traversal without making any changes itself while tExp is defined to immediately call tExp'. Within a Traverser instance, one overloads tExp to implement modifications of an alternative of Exp (for example LetE) and refers to the default tExp' definitions for all other alternatives by also declaring tExp x = tExp' x.

Finally, the function tMain starts the pass by calling tDecs with the monad-specific starting state; it returns the resulting code and additional passes created by the current pass for subsequent execution and wraps up both in Template Haskell's Q monad. The tMain function is called from a main loop described in a following subsection.

3.3.5 Global Actions

Many preprocessing passes only modify the abstract syntax tree very locally. For instance, for a typical let transformation in Haskell only the LetE alternative of the transformation function tExp would have to be overloaded. In the same way state changes are often limited to local actions. On the other hand in the same passes one often needs to insert actions on the internal state which are triggered on *every* part of the syntax tree. In the let transformation one would for instance like to determine *where* each transformation happened; this output of the form:

```
in function 'f',
  in alternative '(TreeNode e l r)',
    in 'case':
      let transformed
```

which might stem from a parsed code fragment like

```
f (TreeLeaf e)     = ...
f (TreeNode e l r) = case e of
                       0: let x = ...
                       ...
```

has to be collected from the place where the let transformation occurred back up to the AST root. This would result in the need to overload almost each transformation function and update the pass state by location information.

To avoid that, we extend the Traverser class as shown in Figure 3.7. As an example, only the extension of tExp for the alternative AppE is shown; all other cases are extended in the same way. Before descending into a branch of a source tree, the function tActionPre is evaluated; after finishing the branch traversal tActionPost is called.

AbsSyn is defined as a collection of Dec, Exp, etc. to avoid having to define tAction functions for each part of abstract syntax. ASTop marks the root of syntax tree, that means the whole program.

```
class (MonadState m s) => Traverser m s where

  -- Functions ------------------------------------------------------
  ...
  tActionPre  :: AbsSyn -> m ()
  tActionPost :: AbsSyn -> m ()

  -- Defaults -------------------------------------------------------
  tActionPre  _ = return ()
  tActionPost _ = return ()

  ...
  tExp' e@(AppE e1 e2) = do tActionPre (AS4 e)
                            e1' <- tExp e1
                            tActionPost (AS4 (AppE e1' e2 ))
                            tActionPre  (AS4 (AppE e1' e2 ))
                            e2' <- tExp e2
                            tActionPost (AS4 (AppE e1' e2'))
                            return (AppE e1' e2')
  ...
```

Figure 3.7: `Traverser` class extended by global actions (excerpt)

```
data AbsSyn = ASTop
            | AS0 Dec   -- not [Dec] but whole Dec branches
            | AS1 Dec
            | AS2 Body
            | AS3 Clause
            | AS4 Exp
            | AS5 Match
            | AS6 Stmt
            | AS7 Range
            | AS8 Type
            deriving Show
```

Both `tActionPre` and `tActionPost` are by default defined to leave the state unaltered, but can be overloaded to house a *set* of global actions which is run before and after entering a node of the syntax tree. A single action is run by `runAction` (see Figure 3.8). As each action usually acts only on one part of the state, this action has to be lifted to the full state by applying the identity function to the remaining state parts. The lifted function can then be used to update the full state. An action set is run by sequencing (using `sequence_`) a list of `runAction` calls in an overloaded version of `tActionPre` or `tActionPost`. In a later section we will show an example on how to attach action sets to a pass.

All this allows us to define global actions separately and to attach selected ones to a preprocessor pass. Predefined actions include: collecting position in code, keeping an indentation level for output, accessing predecessor nodes in the AST, and

```
runAction :: (MonadState m b) =>
             AbsSyn ->                      -- encountered object
             ( AbsSyn   -> (a -> a),        -- action
               (a -> a) -> (b -> b) ) ->    -- lifting
             m ()
runAction x (action, lift) = do update (lift (action x))
                                return ()
```

Figure 3.8: Lift action into full state, run action, update state

```
aPos :: AbsSyn -> (([String],[String]) -> ([String],[String]))
aPos (AS0 _) =
  \(fin,act) -> (fin++act,[])
aPos (AS1 (FunD n cs)) =
  \(fin,act) -> (fin, aPosAdd ("in function "++n++", ") act)
aPos (AS1 (ValD (VarP n) body ds)) =
  \(fin,act) -> (fin, aPosAdd ("in constant "++(To.var n)++", ") act)
aPos (AS1 (ClassD cxt name vars ds)) =
  \(fin,act) -> (fin, aPosAdd ("in class "++name++", ") act)
aPos (AS1 (InstanceD cxt typ ds)) =
  \(fin,act) -> (fin, aPosAdd ("in instance, ") act)
aPos (AS4 (LetE ds e1)) =
  \(fin,act) -> (fin, aPosAdd ("in let, ") act)
aPos _ =
  id

aPosAdd :: String -> [String] -> [String]
aPosAdd m is = [m++" "++i | i <- is, i /= []]
```

Figure 3.9: Position tracking action (use with `tActionPost`)

others. As an example Figure 3.9 shows an action for keeping a position information in the style of the example shown at the beginning of this subsection. To keep the necessary position information, the pass state is extended by

```
([String],[String])
```

A string will represent a position. We need a list of strings, as the event, whose position in the code we want to record, may happen more than one time. The second string list will record the positions for one branch of the syntax tree. If the traversal returns to the tree root, the second list will be appended to the first one which collects the final position results. This way we avoid to get a kind of depth-first traversal. `aPosAdd` adds the current position to all position strings only if the position string is non-empty. The position strings are built on the way up from an event to the syntax tree root, therefore `tActionPost` has to be used.

In total, global actions are advantageous because:

- One can easily assign global actions in an aspect-oriented way. Actions can

be predefined and grouped and applied as sets.

- Using wildcards we can define actions for whole syntactical groups (like all expressions).

3.3.6 The Main Loop

We have seen in the previous subsections how the preprocessor is started and how a preprocessing pass can be coded. But how can we set up and execute a series of passes? We start by defining a list of passes (see Figure 3.10), which in this case contains two fictional passes, namely a pass for the automatic derivation of class instances and a simplification pass.

```
type Passes = [Pass]
data Pass   = forall m s . (Traverser m s) => Pass (m s)

passes :: Passes
passes = [Pass ( return (Derive  ([],[])) :: (ST Derive)   Derive   ),
          Pass ( return (Simplify []    ) :: (ST Simplify) Simplify )]
```

Figure 3.10: Definition of passes with exemplary pass list

Preprocessing is started by a call to `doLoop` (see Figure 3.3 and Figure 3.11) which applies the passes `ps` to the given code `sc` by calling `doPasses`. The total number of performed passes is counted and shown. `doPasses` successively applies each pass by first announcing its start and then calling its specific version of `tMain`.

```
doLoop :: [Dec] -> Passes -> Q [Dec]
doLoop sc ps = do (sc',n) <- doPasses sc 0 ps
                  To.box ((show n) ++ " pass(es) performed.")
                  return sc'

doPasses :: [Dec] -> Int -> Passes -> Q ([Dec], Int)
doPasses sc n []            = return (sc, n)
doPasses sc n ((Pass p):ps) =
  do To.box ("Pass " ++ (show n) ++ ": " ++ (tName p))
     (sc', newps) <- tMain p sc
     doPasses sc' (n+1) (newps ++ ps)
```

Figure 3.11: Loop functions `doLoop` and `doPasses`

New passes created by the current pass are executed right after the the current pass has ended. This could for example be used to tidy up code by invoking a simplification pass after finishing an inlining pass. Note that there is no automatic bail out mechanism, which means that the pass designer has to make sure that the pass list terminates.

3.4 Automatic Instance Derivation

In the language definition of Haskell, the class mechanism is an important construct used for overloading. Datatypes become members of a class by defining a specialised class instance. By demanding additional class memberships before allowing an instance definition, a whole graph of dependent classes is built in the Haskell definition. Analogously, classes are used within Eden to provide overloaded functions for data structure evaluation and data value transmission. Usually instances have to be defined by hand. As instance definitions most often follow regular patterns (which have to be adapted a bit to the specific datatype), this is a tedious task. Therefore there have been some efforts to automate *instance derivation*:

- The Haskell 98 Report already defines a `deriving` construct (Section 4.3.3) which provides automatic instance derivation for the following type classes:

  ```
  Eq, Ord, Enum, Bounded, Show, Read
  ```

  For each class, Chapter 10 of the Report states exactly when and how an instance of that class can be derived for a given data structure. Of course the standard instance derived is not always the one desired; for example, `Show` does not always deliver the kind of output one would like to see and is therefore often coded by hand. If the `deriving` keyword is left out a handmade instance can be given instead.

  A disadvantage is that the `deriving` implementation, at least for the GHC, is hidden in the compiler implementation. Therefore it can neither be extended by new schemes for additional classes nor modified easily.

- *Derive*[216] is a tool for parsing a Haskell program which allows for the definition of Haskell-coded rules which are applied to the parsed program. These rules can then generate additional code which is appended to the original code. *Derive* not only allows for the definition of new derivation schemes, but also reimplements instance derivation of the standard classes mentioned above. It can be used easily by inserting special comments into the Haskell code:

  ```
  {-!for Mytype derive : Eq,Show,Read!-}
  ```

  Rules for instance derivation are given as pairs of a string (containing the name of the rule) and a function mapping an abstract type representation to a text which contains the newly generated code.

- *DrIFT* is the successor to *Derive*. While some features have been improved (small utility functions can now also be generated for a data type), it preserves the basic mechanism shown for *Derive*.

In addition to these tools the preprocessor shown in the last subsection is also capable of deriving instances. One just has to define a pass which scans the given source code for datatype declarations, generates instance code for these types, and adds the code to the original source code. In contrast to the `deriving` construct,

such a preprocessor pass for instance derivation is easily extendable. While *Derive* and *DrIFT* both have to introduce new abstract syntax definitions we can easily use the standardised one of Template Haskell.

In general, one can follow a common pattern when defining a deriving pass:

1. The pass overloads `tDec` for `DataD` (and also for `NewtypeD`) to trigger instance derivation when a data structure declaration is encountered. This means, that the instance is generated for *every* datatype contained in the program and not only for *selected* ones. But this restriction could easily be alleviated by giving a list of data type names, for which instances have to be derived, to the deriving pass.

2. The pass state contains a `[Dec]` entry for collecting the derived code for each instance.

3. When the pass finishes, the collected instance code is appended to the source code.

Obeying that pattern, one can define a general derivation scheme in Haskell which is then applied to every data structure definition in a program to be compiled. As implicitly available data structures like tuples, numbers, strings, `Either`, or `Maybe` are imported and not immediately visible, these will be predefined in Template Haskell Syntax in a separate module and implicitly exposed to instance derivation. For example, the imported definitions for a pair and a list look like this:

```
th_bt_Tup2 :: Dec
th_bt_Tup2 =
  DataD [] "Data.Tuple:(,)" ["a","b"]
        [NormalC "Data.Tuple:(,)" [(NotStrict,VarT "a"),
                                   (NotStrict,VarT "b")]] []

th_bt_List :: Dec
th_bt_List =
  DataD [] "GHC.Base:[]" ["a"]
        [NormalC "GHC.Base:[]" [],
         NormalC "GHC.Base::" [(NotStrict,VarT "a"),
                               (NotStrict,AppT ListT (VarT "a"))]] []
```

The following basic types are described in the same way and thus made implicitly visible to instance derivation:

```
Char, Bool, Int, Integer, Double, Float
Unit
(a,b), (a,b,c), (a,b,c,d), (a,b,c,d,e), (a,b,c,d,e,f)
[a], Maybe a, Either a b
```

One could also think of redefining instance derivation for classes usually covered by GHC, like `Eq` or `Ord`. This does not clash with compiler-internal derivation as the keyword `deriving` can just be left out.

3.4.1 Deriving `NFData`

Let us now turn to the derivation of instances for Eden-specific type classes. We will present a class for demand control, which we will discuss in detail later in Chapter 5. For now we are only interested in how to derive instances for that class automatically.

For controlling process creation order and steering evaluation depth in Eden one frequently needs functions for controlling evaluation of data structures based on the `seq` library function [118] . In Eden as well as in Glasgow Parallel Haskell [205] these functions are united in a type class called `NFData`:

```
class NFData a where
  xi1 :: a -> ()
  xi2 :: a -> ()
```

Functions `xi1` and `xi2` enforce different evaluation depths for a given argument value; the most commonly used `xi2` demands the *spine* of a data structure which means that the recursive structure is forced without touching anything else. Evaluation is demanded via the function `seq :: a -> b -> b` which evaluates its first argument to weak-head normal form and then returns the second argument.

---

2 Example (`xi1` and `xi2` for `[a]`)
The functions `xi1` and `xi2` for the list data structure are:

```
xi1 :: [a] -> ()          -- no traversal, just outermost constructor
xi1 []    = ()
xi1 (_:_) = ()

xi2 :: [a] -> ()          -- traversal of recursive structure
xi2 []      = ()
xi2 (v1:v2) = seq (xi2 v2) ()
```

Instances of this class usually have to be written by hand, a tedious and error-prone task. Figure 3.12 shows a pass which automatically derives instances of `NFData` for every `data` declaration found (and selected base types like tuples) and adds it to the program. To do that, a state for containing the newly generated instance code is defined at first. Using that state, a `Traverser` instance is constructed. The `tMain` function defines an empty starting state and temporarily appends Haskell's basic data types to the source code. Then preprocessing is started by calling `tDecs`. When `tDecs` returns, the additional instance code generated during the traversal is permanently added to the source code (the file is left unaltered) and returned.

Inside the pass, only `tDec` is overloaded for `DataD` (`NewtypeD` is left out in this presentation but can be handled identically). A single state update appears, which appends the newly generated instance code `ds` to the code generated so far. The instance itself is defined as a combination of

- a *class context*, which has to be inferred (described in next Subsection),

```
newtype DeriveNFData = NFData [Dec]
instance Traverser (ST DeriveNFData) DeriveNFData where
  tName _    = "Derive NFData"
  tMain p sc =
    do let startstate           = NFData []
       let sc'                  = sc ++ Ty.th_bt_Comp
       let ST f                 = tDecs sc'
       let (_, NFData endstate) = f startstate
       qIO (putStrLn ("=>"++(show (length endstate))++" NFData derived"))
       return (sc ++ endstate, [])
  tDec d@(DataD c name vars cons ders) =
    do tActionPre (AS1 d)
       update (\(NFData ds_sofar) -> NFData (ds_sofar++ds))
       tActionPost (AS1 d)
       return d
    where
      ds    = [InstanceD (infer_context "NFData" d) iName decs]
      iName = AppT (ConT "NFData") typ
      typ   = foldl (\z v -> AppT z (VarT v)) (ConT name) vars
      decs  = [FunD "xi1" [derive_xi1_cl      con | con <- cons]] ++
              [FunD "xi2" [derive_xi2_cl name con | con <- cons]]

      derive_xi1_cl :: Con -> Clause
      derive_nfdata_xi1_cl (NormalC cName cElems) =
        (Clause [ConP cName pats] (NormalB rhs) [])
        where pats = [WildP | _ <- [1..(length cElems)]]
              rhs  = TupE []

      derive_xi2_cl :: String -> Con -> Clause
      derive_nfdata_xi2_cl name (NormalC cName cElems) =
        (Clause [ConP cName (map VarP cVars)] (NormalB rhs) [])
        where cVars  = zipWith (:) (repeat 'v')
                       [show i | i <- [1..(length cElems)]]
              rhs    = foldr f (TupE []) cVars'
              f v e  = AppE (AppE (VarE "seq")
                                  (AppE (VarE "xi2") (VarE v))) e
              zs     = zip cElems cVars
              cVars' = [v | (t, v) <- zs, find (snd t) == name]

              find   :: Type -> String
              find (ConT n)   = n
              find (AppT a b) = find a
              find _          = ""
  tDec x = tDec' x
```

Figure 3.12: `NFData` instance derivation pass

- its *name*, which is a direct composition of the class name and the data structure which becomes a class member,
- and finally the *function declarations* defining the instance's specific behaviour.

Corresponding to the two functions contained in `NFData`, the declarations `decs` are divided in two parts. For each function, a function alternative (or clause) for each data constructor is generated by calling `derive_xi1_cl` or `derive_xi2_cl`, respectively. Without going into details, each function generates patterns and right-hand sides for each clause. This can be followed by comparing the code with Template Haskell's abstract syntax shown in Figures 3.1 and 3.2.

3.4.2 Context Inference

As soon as parametric polymorphism is used, the *context* of a derived instance is no longer immediately clear. If for example an `Eq` instance has to be derived for `[a]`, then one has to demand the existence of an `Eq` instance for `a` beforehand:

```
instance Eq a => Eq [a] where
  ...
```

This is the case because for comparing elements one has to step back to a customised comparison functions for elements of type `a`. As we derive instances automatically, we also have to derive instance contexts automatically. We do this by

1. scanning the data structure and generating a set of context constraints,
2. successively simplifying the constraint set until a fixed point has been reached.

We will take the following datatype as a running example:

```
data T a b = C1 Int | C2 (T b a) | C3 a b
```

It represents the different cases which are also reflected in the constraint representation of Figure 3.13. Besides the empty constraint `CE` there is a constraint `CB` on base types. `CB` contains two strings which describe the class name and the base type. `CV` describes constraints on variables and finally `CD` describes constraints on polymorphic datatypes. `CD` contains the class name, the type constructor name, and a list of type variable names. A context is then a set of constraints. This context is translated to Template Haskell's context definition via the function `context2cxt`; also, we define a standard `Ord` instance for a constraint. To simplify the comparison of contexts, Figure 3.14 gives the definition of a function for translating a context into a standard canonised form. It removes empty constraints and true constraints, eliminates duplicates, and sorts the context in a predefined order. A constraint is true if the instance demanded by the constraint has already be defined by default. This is the case for all instances contained in the standard prelude, like all combinations of

```
Eq, Ord
```

with

```
data Constraint =
    CE                          -- {}
  | CB String String            -- Eq Int, Eq Char, ...
  | CV String String            -- Eq a, Ord b, ...
  | CD String String [String]   -- Eq T, Eq (T a), Eq (T a b), ...
  deriving (Eq, Show)

type Context = [Constraint]

context2cxt :: Context -> Cxt
context2cxt = map c2t . elimCE
  where c2t :: Constraint -> Type
        c2t (CB c t)    = AppT (ConT c) (ConT t)
        c2t (CV c v)    = AppT (ConT c) (VarT v)
        c2t (CD c t vs) = foldl AppT (ConT c) (map VarT vs)

        elimCE :: Context -> Context
        elimCE cs = [ c | c<-cs, c/=CE ]

instance Ord Constraint where ...
```

Figure 3.13: Constraint and context definition

```
canonise :: Context -> Context
canonise = order . elimDup . elimTrue . elimCE
  where elimCE :: Context -> Context
        elimCE cs = [c | c<-cs, c/=CE]

        elimTrue :: Context -> Context
        elimTrue cs = [c | c<-cs, not (elem c truecontexts)]

        elimDup :: Context -> Context
        elimDup []     = []
        elimDup (c:cs) | elem c cs = elimDup cs
                       | otherwise = c : elimDup cs

        order :: Context -> Context
        order = qsort
```

Figure 3.14: Canonising a context

```
build_equation :: String -> Dec -> (Constraint, Context)
build_equation inst (DataD _ dname namevars cons _) =
  (CD inst dname namevars, concat [c2c c | c <- cons])
  where c2c :: Con -> [Constraint]
        c2c (NormalC _ sts) = [trans1 inst t | (_, t) <- sts]

        trans1 :: String -> Type -> Constraint
        trans1 i (ConT t)              = CB i t
        trans1 i (VarT t)              = CV i t
        trans1 i (AppT t1 t2)          = CD i d vs
          where (d,vs) = trans2 (AppT t1 t2)

        trans2 :: Type -> (String, [String])
        trans2 ListT              = ("GHC.Base:[]", [])
        trans2 (ConT t)           = (t, [])
        trans2 (AppT t1 (VarT t)) = (d, vs++[t])
          where (d, vs) = trans2 t1
        trans2 (AppT t1 (ConT t)) = (d, vs++[t])
          where (d, vs) = trans2 t1
```

Figure 3.15: Building the constraint equation (excerpt)

`Int, Float, Double, Bool, Char`

Figure 3.15 shows the function for building the constraint set out of a given data structure. For every constructor, it generates a set of constraints. This is done by demanding an corresponding instance for every element contained in the constructor. For our example datatype `T` given above it generates the following set:

```
[ CB "Eq" "GHC.Base:Int",                  -- Constructor C1
  CD "Eq" "Main:T" ["b'11","a'10"],        -- Constructor C2
  CV "Eq" "a'10", CV "Eq" "b'11"     ]     -- Constructor C3
```

Canonising will eliminate the true context `Eq Int` and reorder the constraints giving:

```
[ CV "Eq" "a'10", CV "Eq" "b'11",
  CD "Eq" "Main:T" ["b'11","a'10"] ]
```

Figure 3.16 shows the code for minimising the constraint set. A series of simplification steps is applied to the original constraint set. By zipping the set series with a set series shifted by an empty constraint, it can easily be determined if a fixed point has been reached by simply searching for the first pair whose elements are equal.

Within each step, every constraint out of `equation` is matched against the `aim` constraint. Every `CE`, `CB`, or `CV` constraint is just copied; if a `CD` constraint is found and both class and datatype name are equal to the ones of the `aim` constraint, then additional constraints are generated via `substitute`. This is necessary, as

recursive datatypes my contain constructors with flipped elements (like `T b a` in `T a b`). For a complete step, consider again our running example. The `current` set

```
solve_equation :: (Constraint, Context) -> Context
solve_equation (aim, equation) = fix pairs
  where steps = iterate (step aim equation) []
        pairs = zip steps ([CE]:steps)

        fix :: [(Context,Context)] -> Context
        fix ((a,b):cs) | a == b    = a
                       | otherwise = fix cs

step :: Constraint -> Context -> Context -> Context
step aim equation current =
  canonise equation'
  where equation' = concat [ match aim current e | e <- equation ]

        match :: Constraint -> Context -> Constraint -> Context
        match (CD i t abs) cur e@(CD i' t' abs') = if (i==i') && (t==t')
            then substitute cur (zip abs abs')
            else [e]
        match _            _   e                 = [e]

        substitute :: Context -> [(String,String)] -> Context
        substitute []            sub = []
        substitute (CE:as)       sub = substitute as sub
        substitute ((CB c b):as) sub = (CB c b) : substitute as sub
        substitute ((CV c v):as) sub = case lookup v sub of
            Just v' -> (CV c v') : substitute as sub
            Nothing -> (CV c v)  : substitute as sub
        substitute ((CD c t vs):as) sub =
            (CD c t vars) : substitute as sub
        where vars = [case lookup v sub of
                        Just v' -> v'
                        Nothing -> v
                      | v <- vs]
```

Figure 3.16: Solving the constraint equation (excerpt)

of constraints has to reach a fixed point:

```
  ------------------------------------------------------------
  1) aim:      Eq (T a b)
     equation: Eq Int, Eq (T b a), Eq a, Eq b
     current:  []

     match (Eq (T a b)) [] (Eq Int)     -> [Eq Int]
```

```
   match (Eq (T a b)) [] (Eq (T b a))
     substitute [] [(a,b),(b,a)]      -> []
   match (Eq (T a b)) [] (Eq a)       -> [Eq a]
   match (Eq (T a b)) [] (Eq b)       -> [Eq b]

   current': canonise [Eq Int, Eq a, Eq b]
                  -> [Eq a, Eq b]
-------------------------------------------------------------
2) aim:       Eq (T a b)
   equation:  Eq Int, Eq (T b a), Eq a, Eq b
   current:   [Eq a, Eq b]

   match (Eq (T a b)) [Eq a, Eq b] (Eq Int)     -> [Eq Int]
   match (Eq (T a b)) [Eq a, Eq b] (Eq (T b a))
     substitute [Eq a, Eq b] [(a,b),(b,a)]      -> [Eq b, Eq a]
   match (Eq (T a b)) [Eq a, Eq b] (Eq a)       -> [Eq a]
   match (Eq (T a b)) [Eq a, Eq b] (Eq b)       -> [Eq b]

   current': canonise [Eq Int, Eq b, Eq a, Eq a, Eq b]
                  -> [Eq a, Eq b]
-------------------------------------------------------------
3) fixed point reached
-------------------------------------------------------------
```

3.5 Reflection and Adaption

Until now we have been quite conservative when using the preprocessor, because we have only used it to add code to otherwise unaltered source code. But there are also other possible uses which are sometimes classified as follows:

- *Reflection* describes the process of a program altering itself (not only adding to itself) with respect to *inner* circumstances to improve itself, for example in terms of runtime or storage use.
- *Adaption* describes the process of a program altering itself with respect to *outer* circumstances to improve itself in the same way as before.

Both are typical meta-programming applications and can be implemented as preprocessor passes. The following two subsections contain examples for both.

3.5.1 Reflection: Eager Transformation

As an example for reflection we will now describe the *eager transformation*[143] which is important for an effective execution of an Eden program. This transformation seeks and transforms certain kinds of `let` expressions. These contain

top-level process applications which would normally due to lazy evaluation be evaluated quite late; late process creation causes usually a decrease in parallelism. To remedy that, these `let` expressions are transformed so that all top-level process applications are executed on entering the function:

```
let r = p # a              let r' = createProcess p a
in e               =>          r  = deLift r'
                           in r' `seq` e
```

Until now this transformation had been implemented as a transformation of Core syntax embedded into the GHC. By expressing it as a preprocessor pass the code size has been reduced from around 300 lines to less than 30 lines, mostly by saving the effort of rewriting compiler front-end mechanisms.

The pass code is shown in Figure 3.17. The pass state is defined to collect the number of transformations performed as well as the position of these transformations. The first state action is done locally upon encountering a suitable `let`, while the second one is implemented as a global action. As we want to transform `let` expressions, `tExp` is overloaded only for `LetE`. Within the `let`, process applications are identified via the predicate `isPapp` and separated from other expressions via `partition`. The process applications inside the `let` declaration, which are of the form `p # x` are then changed to `createProcess` applications via `toCP`. The `let` expression `e1'` is then changed to `e''` by adding process creation enforcements via `seq`. The internal state is updated to reflect the number of performed transformations and to store their positions before the reconstructed `let` expression is returned.

This is a typical Eden-specific reflection pass. Among others one could also imagine the following ones:

- other *transformational passes*:
  - for each function, pull all process applications (not only the ones contained in a `let`) to the beginning of the function's main expression
  - for each process abstraction, try to inline arguments into the abstraction to save argument transmission when the process is created
- *program analysis passes*:
  - for the whole program, report which functions contain Eden language constructs or skeletons and all remaining functions which are not involved into parallelism
  - for the whole program, analyse whether a dynamic channel is used twice

3.5.2 Adaption: Self-Configuring Skeletons

As an example for adaption, we will shortly sketch the approach taken by Hammond et al.[79] which lets the programmer choose a general type of skeleton in his program but instantiates that skeleton with a selected environment-dependent and cost-minimal implementation of that skeleton automatically.

```
newtype Eager = Eager (Int, ([String],[String]))

instance Traverser (ST Eager) Eager where
  tName _    = "Eager transformation"
  tMain p sc =
    do let startstate = Eager (0, ([],[]))
       let ST f = tDecs sc
       let (sc', Eager (count,(positions,_))) = f startstate
       To.printToScreen positions
       return (sc', [])
  tExp e@(LetE ds e1) =
    do ds' <- mapM tDec ds; e1' <- tExp e1
       let (papps, normals) = partition isPapp ds'
       let vars = [v | (ValD (VarP v) _ _) <- papps]
       let ds'' = normals ++ (concatMap toCP papps)
       let e''  = foldr f e1' [VarE (v++"'") | v <- vars]
       let nET  = length papps
       let vals = ["in "++(To.stripvar v)++", Eager" | v <- vars]
       update (\(Eager (n,    (f,        ps),fs,pr)) ->
                 Eager (n+nET,(f,vals++ps),fs,pr))
       let e' = (LetE ds'' e'')
       tActionPost (AS4 e')
       return e'
    where
      f v r = AppE (AppE (VarE "seq") v) r
      isPapp :: Dec -> Bool
      isPapp (ValD (VarP _) (NormalB (InfixE _ (VarE "EdenSim:#") _)) _)
        = True
      isPapp _ = False

      toCP :: Dec -> [Dec]
      toCP (ValD (VarP o)
             (NormalB (InfixE (Just p) (VarE "EdenSim:#") (Just v)))
            des)
        = [(ValD (VarP (o++"'"))
             (NormalB (AppE (AppE (VarE "createProcess") p) v))
            des),
           (ValD (VarP o)
             (NormalB (AppE (VarE "deLift") (VarE (o++"'"))))
            []) ]
  tExp x             = tExp' x
  tActionPost x      = sequence_ [ runAction x (aPos, liftPos) ]
    where liftPos :: (([String],[String]) -> ([String],[String])) ->
                     (Eager -> Eager)
          liftPos f (Eager (c, ps)) = Eager (c, f ps)
```

Figure 3.17: Eager transformation for Eden

Adaption means, that a program adapts itself according to outer influences. In a parallel setting, what outer factors influencing the performance of a parallel program are there?

- *System-specific* informations include the network topology of the parallel computer, the number of nodes included, and the communication latency.
- *Problem-specific* informations include the size of the data structures involved, the problem-specific data granularity (minimal and maximal independent task sizes), the problem's suitability for different algorithms and its parallelisability.

A generalised parallel method for a certain kind of collaboration between a set of processes, a skeleton, can be seen as a kind of logical pattern whose implementation can vary in many facets. Therefore usually a set of *implementation skeletons*[119] belong to a single logical skeleton. For example, a parallelised map function can be implemented in at least four ways (as shown in our paper[119] and described in Section 6.3): The first way directly reflects the behaviour of map by creating a process for each list element. The second way creates a process for each machine node and distributes the list element over these node processes, saving process creation costs. The third way embeds process argument directly into the process, so that at creation time the arguments are incorporated by the process and need not be transmitted separately. The fourth way is especially suited for varying task sizes induced by the list elements; it creates a worker process on each machine node and dynamically assigns tasks to idle workers.

Normally it's up to the programmer to choose the most suited implementation skeleton after having chosen the general skeleton. This decision is usually made by hand depending on the outer factors shown above and often some intuition. With automatic skeletons[79] , each implementation skeleton comes along with a specific cost model. Given a set of environmental factors like the ones above, the cost for running each skeleton can be calculated statically. Therefore, the most efficient one can be chosen at compile-time. From a programming point of view, instead of using a hand-selected skeleton with hand-selected parameters for granularity control

```
... (myMap myParameters) f xs ...
```

one would now insert a Template Haskell splice `autoMap` which is given actual environmental parameters

```
... $(autoMap environment) f xs ...
```

After calculating the costs of all predefined implementations for `map` the splice will then replace itself by a call for the most efficient one

```
... (bestMap bestParameters) f xs ...
```

Environmental factors can be stored in different ways. One approach taken within Fortress[196] is to map the parallel computer to a hierarchical data structure (see for example Figure 3.18, where each level represents a different level of parallelism associated with a memory and cache model and an inter-level communication latency. Later processes together with data structure partitions can systematically be placed in this hierarchy.

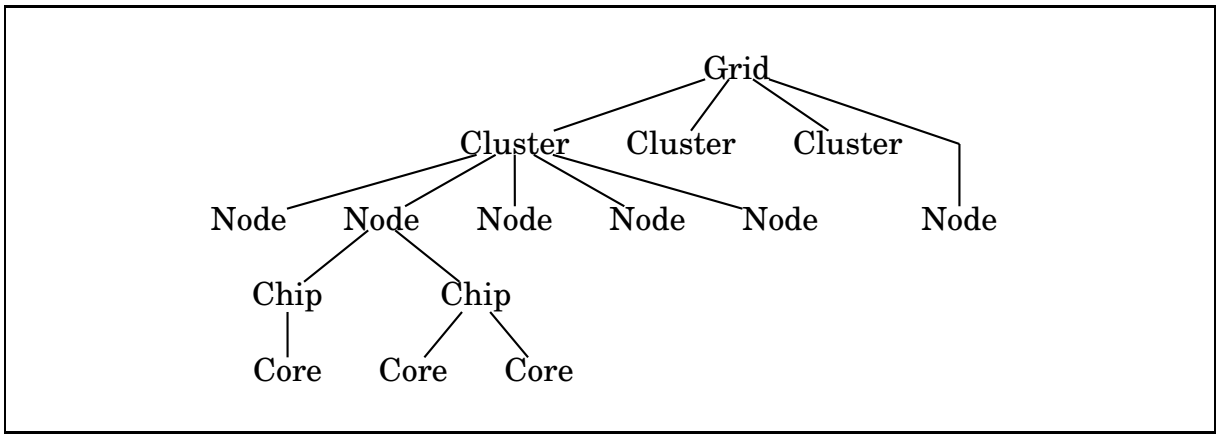


Figure 3.18: Internal parallel computer representation

This scheme can be extended by using the preprocessor developed in this thesis. We can now define a general preprocessing pass which by itself

1. detects all standard skeletons (which are given firm names in the Eden library module) used in a program
2. assumes the same environment description as the one given in Hammond et al. [79]
3. replaces each one by its most efficient relative in the family of implementation skeletons

The advantage of this approach is that the programmer does not need to know about Template Haskell syntax. Furthermore it allows for the easy extension of the former approach to deal with all standard skeletons in a uniform way.

Instead of using specialised implementation skeleton calls which may be (possibly unexpectedly) changed by subsequent preprocessing, one could also decide to let the programmer use more general functions which denote the general skeleton class and serve as placeholders for their most efficient implementations. If, however, specialised implementation skeletons are used by the programmer the preprocessing pass could also act as a non-transforming pass which issues warnings if non-optimal implementation skeletons are detected.

4. Generic Programming for Eden

"Generic programming has ... major advantages over 'one-shot' programming, since genericity makes it possible to write programs that solve a class of problems once and for all ..."

In Section 1.5 "Why Generic Programming?"of Backhouse et al.[12]

4.1 Motivation

Sequentially, one of the most often used abstractions in Haskell programs is surely *parametric polymorphism*; this mechanism, if semantically applicable, saves us the effort of having to write different implementations of a function for different element types. As an example, the function `elem` on lists would normally have to be implemented for every possible element type:

```
elemI :: Int    -> [Int]    -> Bool
elemI e []     = False
elemI e (x:xs) | x == e    = True
               | otherwise = elemI e xs

elemC :: Char   -> [Char]   -> Bool
elemM :: Matrix -> [Matrix] -> Bool
...
```

Although determining whether the element is contained within the structure is a simple structure traversal and independent of the element datatype (given an equality check), a separate function would have to be defined for each element type. Haskell's polymorphic typesystem now allows for the introduction of type variables to capture the basic similarity of these functions:

```
elem :: Eq a => a -> [a] -> Bool
elem e []     = False
elem e (x:xs) | x == e    = True
              | otherwise = elem e xs
```

This polymorphic version can be used with all element types which provide an equality check, which is already quite general. Nevertheless further abstraction is desirable and possible, as `elem` can only operate on lists. If other data structures

around the type variable `a` would have to be searched, the same situation as before would arise; we would have to define separate implementations although the basic search procedure remains unchanged:

```
elemList    :: Eq a => a -> [a]       -> Bool
elemTree    :: Eq a => a -> Tree a    -> Bool
elemHMQueue :: Eq a => a -> HMQueue a -> Bool
...
```

What we need is another generalisation which makes it possible to summarise these function in a single one. Via *constructor classes*, Haskell again leads the way to a solution, because using them we can let type constructors vary:

```
elemGeneric :: Eq a => forall t. forall a. a -> t a -> Bool
```

Given an implementation for `elemGeneric` (which is not trivial and the topic of this chapter), we have defined once and for all the `elem` check for all data structures of kind `(* -> *)` and all element types. When abstracting like that one also speaks of *structural polymorphism* [180] , *polytypic functions* [111] , or in a more general context of *generic programming*. For more information on approaches to generic programming see Section 4.4 (for general approaches) and Section 9.2 (for specific implementations). Providing an implementation for a generic function like `elemGeneric` above, is not trivial, as it is not immediately clear how a function can be defined without knowing its argument datatypes completely; classical pattern-matching is then impossible. In this chapter we will show how to implement an approach to generic programming which can not only be implemented with means provided mostly by Haskell but which is also suited for parallel programming.

We are interested in efficiently programming parallel computers in Eden. Generalising a function will not make it run faster. In the contrary, the additional overhead can even slow it down. Then why do we consider genericity useful for parallel functional programming? We do because it fits well into the whole picture:

1. Even if we do not apply generic methods to Eden's parallel constructs, they will be useful on a purely sequential level. All advantages in favour of usual generic programming therefore also apply to generic programming in Eden.

2. Parallelism can be seen as an occasion for simultaneity induced by only small or even no dependences between expressions which are exploited to a certain amount for increasing program performance. In Eden, parallelism can be reflected on the program-level in two ways:

   - as *control parallelism*: process cooperation patterns reveal independence; these are generalised via skeletons/HOFs which means that parameter functions may vary
   - as *data parallelism*: data structure organisations reveal independence; these are generalised via parametric and now structural polymorphism which means that types may vary

This means, that just as abstract control parallelism is reflected by skeletons, abstract data parallelism is reflected by the combination of parametric and structural polymorphism. This dualism will be further discussed in Chapters 6 and 7.

3. In addition to data parallelism, parallelism in general depends very much on data structure handling (partitioning, granularity control), which can greatly benefit from generic functions.

4. Although generic functions will not speed up a program, they will speed up the implementation process. The high-level programming style eases later code changes and supports speed gains by specialising generic functions to more efficient versions. Therefore generic functions advocate to rapidly prototype a parallel program suitable for later tuning.

For direct combination with parallel programming, generic methods will usually be applied in a *structure-oriented* way. This means, that for parallelism usually large structures and not small values are processed. This is the case, as parallel programming mostly deals with partitioning, merging, traversing, searching, zipping, mapping, and unifying structures. When being combined with generic methods the focus will therefore be more on polymorphic structures and not so much on the usual base types like `Int` or `Char` contained within them. Generic parallel functions are therefore more functions for programming 'in the large' than functions for changing small parts of a structure. Therefore our approach to generic programming is tailored for being applied in the context of a parallel functional programming language like Eden.

> Section 4.2 introduces into generic programming by giving a short description of data structures in general. In the same way, Section 4.3 summarises Haskell's type system together with a couple of implementation-dependent extensions. Then Section 4.4 shortly sketches the different approaches to implementing generic functionality so far. Subsequently, Section 4.5 in depth describes a new approach to generic programming geared towards being used in the parallel Haskell dialect Eden. To flesh out the approach, Section 4.6 shows generic versions of well-known sequential standard Haskell functions.
>
> The chapter is based on our technical report[174] .

4.2 Data Structures

To prepare the ground for the oncoming sections on generic programming, we will now set out to give a short summary on data structures. These can be described via methods of algebraic specification. These methods are well-known[107, 60] ; we will, however, give a more practical description.

When a data structure is needed, one usually starts not by defining the structure itself, but by defining the set of functions wanted for accessing and manipulating the structure. The resulting *abstract datatype* can then only be accessed via

these functions while its interior implementation is invisible. This *abstraction barrier* [17] is useful, because the concrete implementation of the data structure can be changed while its interface is left untouched. A data structure is then typically accessed via a set of functions tailored for selecting special parts of the structure. Such an abstract datatype spans a *space of different concrete implementations*. For example, a queue with its typical interface can be implemented in different ways: via a simple list with expensive access, or as a Hood-Melville queue [92] which breaks the queue into two lists providing more efficiency when accessing both ends of the queue. A concrete implementation can be used on its own without further access functions, as its parts are known.

Within Haskell both approaches are possible: One can construct an abstract datatype by creating a module which exports only the functions needed for operating the data structure while hiding the concrete implementation. But more often datatypes are defined explicitly and reveal their interior to the outside, which enables the programmer to avoid access operations and to use the more popular pattern-matching style of programming.

Besides predefined datatypes Haskell allows for the definition of user-defined datatypes; for doing so usually the `data` declaration is used:

$$\begin{array}{lcl} data\ C \Rightarrow\ T\ a_1 \ldots a_n & = & D_1\ t_{11}\ ...\ t_{1m_1} \\ & | & D_2\ t_{21}\ ...\ t_{2m_2} \\ & & \vdots \\ & | & D_k\ t_{k1}\ ...\ t_{km_k} \end{array}$$

This declaration introduces a new datatype with a *type constructor* T which is parameterised via the *type variables* $a_1, \ldots, a_n$. On the value side, new *data constructors* $D_1, \ldots, D_k$ are defined. Data constructors can also be seen as functions and be treated as such; especially, partial application of argument values is valid. The whole type is restricted by a *type context* C which can demand the existence of class instances for types assigned to type variables $a_1, \ldots, a_n$. Type synonyms can be defined via the `type` declaration; this means, that a type is also known under another type constructor without introducing new data constructors. The same can be done via *newtype* but with the difference that a new data constructor is introduced. The compositional building of values out of nested data constructors is accompanied by the dual concept of *pattern-matching* for value decomposition in function definitions.

Data constructors contain (possibly empty) sequences of elements. Different combinations can arise:

```
data Ex1     = A1                              -- empty constructor
             | A2 Int Char                     -- base types
data Ex2 a   = B1 a                            -- type vars
             | B2 Int (Ex2 a) (Ex2 a)          -- recursion
data Ex3 a b = C1 (Ex3 a b) (Ex3 b a)          -- exchanged type vars
             | C2 (Ex3 (Ex3 a b) b)            -- exponential type
             | C3 a b
```

The first two examples show the most common data constructor layouts, while the third one shows the (quite uncommon) flipping of elements and exponential (or

non-uniform[161]) datatypes. When designing an approach to generic programming one has to deal with these cases.

4.3 The Type System in a Nutshell

In this section we will shortly describe Haskell's type system and some of its extensions which are not defined within the Haskell language report. Some parts of the following work is based on the mechanisms shown here. Like in the previous section we will give an application- and language-oriented description of the topic and skip the large field of type theory, as it is not needed here.

Haskell is all about *values*: These are nested applications of various data constructors which we have introduced in the previous section. Beneath these, also functions are first-class values in Haskell. Data constructors to which not all elements are applied are also functions. To avoid the incorrect application of values within a functional program, families of values are introduced and called *types* (see Pierce[168] or Chapter 8 in van Leeuwen[208]). Haskell incorporates a *Hindley-Milner type system* [154] which allows for the automatic and static inference of the most general type for every function and expression. Explicitly given types are not only checked for correctness, but missing types are also inferred. Many errors in a program then show up during static type inference instead of yielding a runtime error, supporting Robin Milner's famous slogan that 'well-typed programs never go wrong'.

Types can themselves be grouped and divided into different *kinds*. Unlike types, kinds are entirely invisible on the language-level and just serve the purpose of checking the validity of type expressions via kind inference. In contrast to only *monomorphic* functions like

```
and :: [Bool] -> Bool
```

the most general type of *polymorphic* functions permits a certain variety in the type parameters. This is reflected by *type variables*:

```
length :: forall a. [a] -> Int
```

The `forall` declaration is implicitly present and can be omitted; this represents the typical Haskell 98 *rank-1 polymorphism*. Kinds then reflect polymorphism by categorising types via their parameters in a curried way by using `(*)` for nullary type constructors and `(->)` for connecting two kinds. The example datatypes `Ex1` to `Ex3` shown in the previous section are of the kinds `(*)`, `(* -> *)`, and `(* -> * -> *)`. These just reflect the number of type variables and are therefore called *first-order* or *flat* kinds. Higher-order kinds arise if type variables are used as type constructors; this will be discussed below.

If we allow for a deeper nesting of the `forall` clause in a type, we obtain *rank-n polymorphism*. This way, were are allowed to hide additional polymorphism inside a type:

```
data List = Cons (forall a. Eq a => a -> a -> Bool) List
          | Nil
```

Parametric polymorphism is powerful but can also be problematic. The simple addition operator `(+)` should and can for example neither be typed monomorphically as `Int -> Int -> Int` (would be too restrictive) nor fully polymorphically as `a -> a -> a` (would be too general as for example functions cannot be added). *Ad-hoc polymorphism* [214, 77, 166] extends the Hindley-Milner type system by defining an intermediate stage which restricts type variables in parametric polymorphism. *Type classes* incorporate these restrictions: Each class contains a set of functions. If these are defined for a datatype (forming an instance), that datatype is allowed to instantiate the restricted type variable. Arithmetic is captured by the `Num` class, so our example would now be specified via

```
(+) :: Num a => a -> a -> a
```

If more than one instance is defined for a class the functions of the class are *overloaded*. In Haskell this mechanism resulted in large graph of interdependent type classes for expressing various restrictions ranging from comparisons to enumerations and numbers. Even multi-parameter classes are allowed (example taken from the GHC User's Guide):

```
class Collection c a where
  union :: c a -> c a -> c a
  ...
```

Given the polymorphic definition of `length` above, one is tempted to generalise over the type constructor to obtain:

```
size :: forall t. forall a. t a -> Int
```

Types like these are possible via *higher-order polymorphism* [113] which induces higher-order kinds. Type correctness can then be determined by inferring its kind which in the end has to be `(*)`. Practically, functions with types like the one above hardly exist, because without special additional mechanisms a function cannot cope with anonymous data constructors.

The combination of ad-hoc-polymorphism and higher-order kinds provides a solution: In contrast to normal type classes *constructor classes* take partially applied type constructors as arguments. A standard example from the Haskell prelude is the `map` functor:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Within an instance of `Functor`, the concrete allocation of `f` is clear; therefore, pattern-matching can be used:

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

4.4 Basic Approaches to Generic Programming

This section explains two basic approaches to implementing generic functionality and discusses their relationship. Based on both approaches, we will present a third one for Eden in the following section.

4.4.1 Static

The *static* approach to generic programming can be summarised as follows. By using constructor classes which can also vary over type constructors one can simply overload the function to be generalised with a specific version for every data structure which might be applied to the function. Each instance can be written statically in full knowledge of the respective data structure and its data constructors. At runtime the correct version will be chosen according to its argument. One could then express a generic length function via the class `Countable`:

```
class Countable t where
  len :: t a -> Int

data BinT a = Leaf a
            | Node a (BinT a) (BinT a)

instance Countable [] where
  len []     = 0
  len (_:xs) = 1 + len xs

instance Countable BinT where
  len (Leaf _)       = 1
  len (Node _ le ri) = 1 + len le + len ri
```

Unpleasantly, one would have to write many instances of the generic function which would therefore be only generic in use, not in definition. This could be automated to some extent in the form of a static deriving mechanism, which would shift the generic function definition from a high to a lower abstraction level (by giving general instructions on how to derive proper instances).

The types shown above differ in their *kind*. While types categorise values, kinds categorise types in terms of their type variables. *Flat* kinds just reflect the arities of type constructors in a curried way. The above example datatypes are of flat kinds `(*)` and `(* -> *)`. (Higher order non-flat kinds arise if type variables are used as type constructors.) One would therefore also need a class for every flat kind, and all that again for various other generic functions resulting in large type contexts, e.g.:

```
class Countable2 t a b   where ...
class Countable3 t a b c where ...
class Zippable2  t a b   where ...
class Zippable3  t a b c where ...
```

```
f :: (Countable2 ..., Zippable2 ..., ...) => ...
```

The main advantage of this approach is: It is fast and simple because no lifting and no compiler or language extension is needed.

4.4.2 Dynamic

The *dynamic* approach to generic programming consists in transferring argument data structures at runtime into a uniform algebraic sum-of-products representation. For example, the binary tree `(BinT a)` shown above would be interpreted as

```
(Leaf * a) + (Node * a * (BinT a) * (BinT a))
```

The generic function, which has to be defined only once on the program level, is designed for working only on such representations. A typical compositional function definition looks like this:

```
genlength<a> :: t a -> Int
genlength<a>   x       = 1
genlength<1>   ()      = 0
genlength<Int> n       = 0
-- analogously for Char,Bool,...
genlength<a+b> (Inl x) = genlength<a> x
genlength<a+b> (Inr y) = genlength<b> y
genlength<a*b> (x, y)  = genlength<a> x +
                         genlength<b> y
```

The generic function will dynamically lift the whole value to the general sum-of-products representation, compute the result by traversing the sum-of-products representation, and for non-basic result types delift it back. Disadvantages include: A language extension has to be implemented, expensive whole-value lifting happens, and functions for lifting and delifting have to be generated. Advantageous is the absence of type contexts and that generic functions are generic in definition and in use, keeping it on a high level of abstraction.

4.5 Generic Programming for Eden

We will now present an approach to introducing generic methods into Eden. Our approach is designed to meet three aims:

- *Based on Haskell to support portability.*
  When implementing supersets of common programming languages, for example an extension of Haskell, it is desirable to implement the superset on top of a mature parent language compiler. As already argued in our preprocessing paper[172] and in Chapter 3, it is even more desirable to implement the superset in the parent language without having to modify the existing compiler.

  Our approach to generic programming is implemented in Haskell with almost no internal parent compiler modification. Therefore it can be much more eas-

ily ported to newer versions of the parent language compiler. This is very important for extensions of Haskell, as Haskell compilers develop quickly; versions have to be followed for being able to make expressive comparisons between parallel Haskell dialects.

- *Combines static and dynamic approach for easy implementation.*
  We combine parts of the static and the dynamic approach: Like in the static approach a class mechanism with statically generated instances will be used to provide ease of implementation. At the same time, generic functions will be definable on the language-level like in the dynamic approach.

- *Structure-oriented to suit parallelism.*
  In addition to being portable, our approach is more directed towards being used together with *parallel* functional languages, as it provides stepwise access to nested data structures. This is important when dealing with parallelism, as the decomposition of algorithms into tasks which are executable in parallel is closely accompanied by the decomposition of nested data structures in collections of substructures. Therefore generic functions in a parallel world are not aimed at skipping uninteresting outer layers of a data structure but are expected to allow access to every layer of a nested data structure for detailed data structure de- and recomposition.

The last point is where we will start: How to gain access to the constructors of an unknown data structure passed as an argument to a generic function.

4.5.1 Gaining Access to Data Structures

As generic functions are expected to work on arbitrary data structures, one faces the problem of defining a function which can cope with arguments of unknown types. But how can we express that? Imagine a function `f` receiving an argument of type `t a`:

```
f :: t a -> ...
f ? = ...
```

Normally one would now define `f` via a couple of alternatives, each of which reflecting a data constructor case of the argument type. But as the exact type constructor `t` is unknown and varies, we cannot know statically which data constructors we will face. Another possibility is now to defer pattern-matching by using somehow generalised selection functions (similar to the specialised `head` and `tail` for lists) after matching against variables:

```
f :: t a -> ...
f x = ... (select_element x) ...
```

This of course only shifts the same problem to a second function. But it leads the way to a solution: We can relocate the deconstruction of an argument value to the place where its parts are needed. If a common deconstruction function is defined for all data structures which may be encountered as arguments, we can deconstruct the given value, work on its internal parts in the way needed, and reconstruct the

value using a reconstruction function also provided by the deconstruction function. This way, we implement a kind of postponed pattern-matching and abstract from data constructors. Such a deconstruction function would then have the type

```
h :: t a -> ( ([a], [t a]),           -- deconstruction
              ([a], [t a]) -> t a )   -- reconstruction
```

which, given a value of kind `(* -> *)`, yields lists of the value's internal parts (a *deconstruction*) together with a complementary *reconstruction* function. Using that deconstruction function, a generic *map* function for data structures with a single type variable (thus called `map1`) can simply be expressed as:

```
map1 :: (a -> b) -> t a -> t b
map1 f xs = reconstruction (as', tas')
  where (deconstruction@(as, tas), reconstruction) = h xs
        as'                                        = map f as
        tas'                                       = map (map1 f) tas
```

To get the well known map for lists, one would use:

```
h :: [a] -> ( ([a],[[a]]), ([a],[[a]])->[a] )
h []     = ( ([], []),   \([], [])   -> []   )
h (x:xs) = ( ([x],[xs]), \([y],[ys]) -> y:ys )
```

A map working on rose trees

```
data Rose a = R a [Rose a]
```

can be obtained by:

```
h :: Rose a -> ( ([a],[Rose a]),
                 ([a],[Rose a]) -> Rose a )
h (R a xs) = ( ([a],[xs]), \([b],[ys]) -> R b ys )
```

Therefore the function `h` provides a wrapping for data structures forming a general interface to its constructors. If defined for all data structures occurring in a program, dynamic generic functions can be defined (like `map1` above) and used. This construction lends itself to overloading via constructor classes. We therefore use a set of constructor classes (one for each of the most common flat kinds), which provide the `h` function for its instances. Static derivation of an instance for the fitting class is done automatically for each data type in the program by a meta-programming mechanism already presented in Chapter 3. Generic functions are then defined only once on the program level using these abstraction functions for being able to work on all data types occurring in the compiled program. This way no compiler or language extension is necessary.

4.5.2 Nested Data Structures

One decision has yet to be made regarding nested data structures: Given the common map function on lists

```
map  :: (a -> b) -> [a] -> [b]
```

what would one in general semantically expect from a generic version of map? Certainly, one would want it to work on arbitrary type constructors `t` instead of only on lists:

```
map1 :: (a -> b) -> t a -> t b
```

But what does that mean when the function is applied to a data structure nested via the replacement of type variables like for example

```
type IntTreeListList = [[BinT Int]]
```

Do we want

α) `a` to match only the *innermost type* `Int` and `t` to match the remaining hull `[[BinT ...]]` or do we want

β) `t` to match only the *outermost type constructor* and `a` to match the remaining `[BinT Int]`?

As map applies a function `(a -> b)`, this decision determines the whole function behaviour: Innermost matching corresponds to working on a carefully selected part of the data type while ignoring the remaining structure and saving the traversal effort; in the case of data structures having a larger flat kind than `(* -> *)`, other mechanisms for choosing one of the innermost types are needed. Outermost matching in a sense generically peels off the outermost layer of a data type and, in the case of nested data structures, may require multiple applications to propagate down the nesting.

α) Innermost matching is explored by the *boilerplate approach* [123] . Within this approach map acts *element-oriented*: map is defined as a generic traversal over arbitrary data structures during which the argument function is applied to every data structure element. Using a special wrapping, the argument function is triggered only if a predefined element type is encountered. The generic map is defined via overloading and reflects its very general applicability (limited by the existence of class instances):

```
class Typeable a => Term a where
  gmapT :: (forall b. Term b => b -> b) -> a -> a

everywhere :: Term a =>
              (forall b. b -> b) -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

The absence of type constructors shows how structure is left aside. Structure traversal is considered as tiresome boilerplate code and abstracted away.

β) In our approach we will use outermost matching. It is *structure-oriented*, as de- and reconstruction clearly reveal the layout of the data structure to the generic function and not only its elements. Then it is easy to write generic functions which not only traverse but also transform data structures. This is especially important for parallel programming, which rests upon data-parallel functions and functions for partitioning large data structures into

collections of smaller parts. In a sense, for organising a parallel computation structure is more important than the elements contained in the structure. Due to its different perspective, our approach results in a different style of generic programming.

We can operate generically on a nested datatype in an outermost fashion by using a corresponding nesting of generic functions: For example, we can apply a function `f :: a -> b` to a nested structure `x :: t1 (t2 a)` by using `map1 (map1 f) x`.

4.5.3 The Abstraction Classes

For implementing our structure-oriented approach, we need to have a generally applicable version of the `h` function shown above. We will now use overloading via type classes to provide this function for different flat kinds. Figure 4.1 shows abstraction constructor classes for the three most common flat kinds `(*)`, `(* -> *)`, and `(* -> * -> *)`. Inside an `Abs` class a set of core functions is defined on top of which external generic functions are constructed later. As it is often helpful to

```
-- at least kind (*) ------------------------------------------------
class Abs0 t where
  null0 :: Maybe t
  suco0 :: t ->                        -- value
           (String,                    -- constructor name
             ( [t],                    -- deconstruction
               [t] -> t )              -- reconstructor
           )

-- at least kind (* -> *) -------------------------------------------
class Abs1 t where
  null1 :: Maybe (t a)
  suco1 :: t a ->                      -- value
           (String,                    -- constructor name
             ( ([a], [t a]),           -- deconstruction
               ([b], [t b]) -> t b )   -- reconstructor
           )

-- at least kind (* -> * -> *) --------------------------------------
class Abs2 t where
  null2 :: Maybe (t a b)
  suco2 :: t a b ->                    -- (analogous)
           (String,
             ( ([a], [b], [t a a], [t a b], [t b a], [t b b]),
               ([c], [d], [t c c], [t c d], [t d c], [t d d]) -> t c d )
           )
```

Figure 4.1: The `Abs0`, `Abs1`, and `Abs2` constructor classes

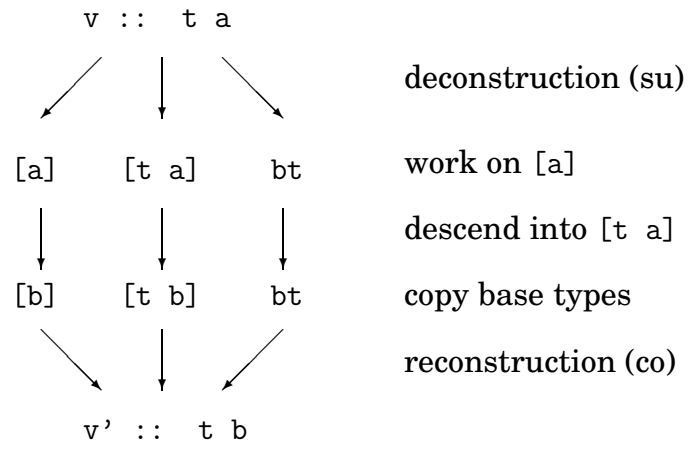


Figure 4.2: De- and reconstruction for kind `(* -> *)`

be able to determine whether a nullary constructor is available, a `null` function is included. Because not every data type has a nullary constructor (for example `Either`), the result is wrapped in `Maybe`. The main abstraction function (formerly called `h`) however is called `suco`, because for a given value it yields the name of the value's constructor, a deconstruction (*su*-btrees) of the value, and a reconstruction function (*co*-mpose) for putting all parts together again (see Figure 4.2). Because the order of type variables can be changed, `suco` has to deal with all possible permutations of type variables. The tuples for deconstruction grow larger accordingly. Fortunately flat kinds higher than `(* -> * -> *)` are rare in average programs.

For a base type like `Char` one would write:

```
instance Abs0 Char where
  null0   = Just 'a'
  suco0 c = ([c],  ([], \[] -> c))
```

A simple data type describing terms of the lambda calculus could be treated via:

```
data LamTerm = Var String
             | App LamTerm LamTerm
             | Abs String  LamTerm

instance Abs0 LamTerm where
  null0             = Nothing
  suco0 (Var s)     = ("Var", ([],      \[]         -> Var s))
  suco0 (App l1 l2) = ("App", ([l1,l2], \[l1',l2'] -> App l1' l2'))
  suco0 (Abs s l)   = ("Abs", ([l],     \[l']       -> Abs s l'))
```

A bit more interesting is the instance for a data type of higher kind:

```
instance Abs1 BinTree where
  null1              = Nothing
  suco1 (Leaf x)     = ("L", (([x],[]),    \([y],[])    -> Leaf y))
  suco1 (Node x l r) = ("N", (([x],[l,r]), \([y],[m,s]) -> Node y m s))
```

The de- as well as the reconstruction actually have to take care of all possible

components which can appear within a data type. But it is advisable to restrict ourselves to such components which can be constructed out of the type constructor and type variables. The examples as well as Figure 4.1 show the components taken care of by these classes, which comprise all type variables and all n-ary combinations of $t\ a_1 \ldots a_n$ because type variables may be flipped like for example:

```
data Flip a b = Foo (Flip a b) (Flip b a) | Bar a b

instance Abs2 Flip where
  null2 = Nothing
  suco2 (Foo f1 f2) = ("Foo",(([],[],[],[f1],[f2],[]),
                      \([],[],[],[g1],[g2],[]) -> Foo g1 g2))
  suco2 (Bar x1 x2) = ("Bar",(([x1],[x2],[],[],[],[]),
                      \([y1],[y2],[],[],[],[]) -> Bar y1 y2))
```

Base types like `Int` or `Char` are copied and are therefore not at the generic function's disposal. If base types were included, one would get into trouble for example with `zipWith`, which would need a list of functions for combining values of any base type. Together with values of base types, all values not directly belonging to the inductive definition of the datatype are also not accessible and therefore usually just copied. This includes the `(List a)` value in `Example`, a case which is covered by other (inductive) approaches to generic programming (like PolyP[112]):

```
data Example a = E1 a (List a) | E2 (Example a)
```

Each class can have instances for types of *at least* the given kind, because one can write instances of `Abs0` not only for `Int` or `Char` but also for `BinTree a` or `Either a b`. This means that one can choose how many type variables are pushed into the type constructor `t` and how many are revealed to the abstraction functions, but only in the order defined by the data type. Of course, an instance for `Abs0` can then be given for every data type. One has to be aware that the decision on how many type variables are exposed to an abstraction class directly influences the semantics of the generic functions derived. Type variables bound to the type constructor are not available to the generic function. Typically one would choose the highest possible class. In a sense one can speak of a 'kind waterfall' as a type of a certain kind can be an instance in its own and all classes of lower kind. For example:

```
instance Abs2  Either       ...   -- Either
instance Abs1 (Either a)    ...
instance Abs0 (Either a b)  ...

instance Abs1 BinTree       ...   -- BinTree
instance Abs0 (BinTree a)   ...
```

Compared to the static approach it is advantageous to abstract from having a class for every generic function. Because many generic functions share the same abstraction functions we can keep type contexts small. Nevertheless for each data type involved in a generic function at least one `Abs` instance will occur.

The instances above show that their derivation is tedious, but not difficult. In the next subsection we show how to generate these instances automatically by means of meta-programming already discussed in Chapter 3.

4.5.4 Implementation

Now we will give a panoramic view across the implementation: When a program containing generic functions is compiled, a preprocessing pass for every kind covered by the abstraction classes is applied to the program. Each pass is defined for the preprocessing mechanism presented in Chapter 3 and is based on metaprogramming via Template Haskell [189] . In essence Template Haskell parses the program, applies a set of preprocessing passes to the resulting internal syntax tree, and writes the program back for further compilation. In our case, each `Abs` pass will scan the input program for `data` declarations, check whether they fit its kind, and derive the suitable instances if this is the case. For each data type of flat kind n, an `Abs` instance is derived for every flat kind equal to or lower than n. Instances for indirectly present base types (like tuples) are also derived. A generic function library based on these abstraction functions is defined normally on the program level and is simply imported by the user's program to provide a predefined collection of generic functions.

Figure 4.3 shows the pass for generating `Abs0` instances. `DeriveAbs0` is the state maintained throughout the syntax tree traversal. Its only component is a list of declarations containing all instances created via the traversal. `tMain` steers the derivation. At first the syntax tree `sc` is extended by declarations of compiler-internal base types like `Bool`, `Maybe`, `Either`, tuples, and so on. These are otherwise implicitly present but now need to be directly available to be considered by the passes. The main machinery is started by the call to `tDecs` which transforms the program. After transformation the state (containing the created `Abs0` instances) is given back and appended to the syntax tree.

`tDecs` itself calls `tDec` for every declaration. `tDec` is the identity function for every declaration with `DataD` being the only exception. The state is extended by the instance for the encountered data type. The instance itself is composed by its name, its type context, and its definitions of `null0` and `suco0`. Both are derived by traversal of the data type constructors of type `[Con]`. Each function generates for each constructor a `Clause` which is a right hand side of a function. `null0` checks whether nullary constructors are available and if this is the case chooses the first one as default. `suco0` scans the data type alternatives and builds the correct de- and reconstruction expressions.

The derivation schemes for `Abs1` and `Abs2` are essentially of the same structure, a bit more complex, much longer, and therefore not shown here.

We will now turn to see how this generic approach can be used in practice.

4.6 Sequential Genericity

Now we will generalise some of the well-known Haskell Prelude functions (see Section 8 in the Haskell language definition[164]). But wait: What does it mean to generalise a function for every datatype? Is there always a unique generalisation, a clear meaning simply extendable to any data type?

Of course not. Many functions of which we seem to have a clear semantic con-

```
newtype DeriveAbs0 = Abs0 [Dec]

instance Traverser (ST DeriveAbs0) DeriveAbs0 where
  tMain p sc = do let startstate         = Abs0 []
                  let sc'                = sc ++ basetypes
                  let ST f               = tDecs sc'
                  let (_, Abs0 endstate) = f startstate
                  return (sc ++ endstate, [])
  tDec d@(DataD c name vars cons ders) =
    do update_state state       -- No kind check necessary,
       return d                 -- Abs0 is derived for every DS
    where
      state = \(Abs0 ds_sofar) -> Abs0 (ds_sofar++ds)
      ds    = [InstanceD [] iName decs]
      iName = AppT (ConT "Abs0")
              (foldl (\z v -> AppT z (VarT v)) (ConT name) vars)
      decs  = [FunD "null0" [derive_null0 cons]             ] ++
              [FunD "suco0" [derive_suco0 con | con <- cons]]

      derive_null0 :: [Con] -> Clause
      derive_null0 cs = Clause [] (NormalB e) []
        where e  = case length nk of
                     0         -> ConE "Data.Maybe:Nothing"
                     otherwise -> AppE (ConE "Data.Maybe:Just")
                                       (ConE (head nk))
              nk = [name c | c <- cs, nullary c]
              nullary :: Con -> Bool
              name    :: Con -> String

      derive_suco0 :: Con -> Clause
      derive_suco0 (NormalC cName cElems) =
        Clause [ConP cName (map VarP cVars)] (NormalB rhs) []
        where
          cVars = zipWith (:) (repeat 'v') (map show [1..length cElems])
          rhs   = TupE [LitE (StringL cName), rest]
          rest  = TupE [decon,recon]
          vs    = [v |((_,t),v) <- zip cElems cVars, t == ConT name]
          vs'   = map (\v -> v ++ "'") vs
          decon = ListE (map VarE vs)
          recon = LamE (map VarP vs') rexp
          rexp  = foldl (\z v -> AppE z (VarE v)) (ConE cName) rvs
          rvs   = [if t == ConT name then v++"'" else v |
                   ((_,t),v) <- zip cElems cVars]
```

Figure 4.3: Derivation scheme for `Abs0` (kind `(*)`) constructor class (excerpt)

```
sf :: (a,(b,c)) -> b        ss :: (a,(b,c)) -> c
sf = fst . snd              ss = snd . snd

dfs0, bfs0, tail0 :: Abs0 t => t -> [t]
dfs0 t = t : concatMap dfs0 (sf (suco0 t))
bfs0   = concat . l
  where l t = [t] : foldr cat []
                    (map l (sf (suco0 t)))
        cat   = zipWith (++)
tail0  = sf . suco0

depth0, size0 :: Abs0 t => t -> Int
depth0 = (+1) . foldl max 0 . map depth0 . tail0
size0  = (+1) . sum .         map size0  . tail0

init0, paths0 :: Abs0 t => t -> [[t]]
init0  t = [init p | p <- paths0 t]
paths0 t | null bs   = [ [t] ]
         | otherwise = [ t:p | b <- bs, p <- paths0 b]
           where bs = sf (suco0 t)
```

Figure 4.4: Sequential generic functions `(*)`, based on `Abs0`

ception are hard to put to general terms. Take `length` as an example: How do you define `length` for `Bool` or `(BinTree a)`? Within the tree, one would normally count occurrences of values of type `a` in the tree. Therefore we chose to generalise only in terms of type variables and recursive occurrences of the datatype itself. Base types are left untouched. Only those functions which work on these type variables or the recursive occurrence are considered for generalisation. These are anyway the most abstract and therefore most interesting ones. For higher kinds, possibly lists, sets, or pairs of results are produced because of the wealth of possible combinations.

4.6.1 Functions for `Abs0`

Figure 4.4 shows a couple of generic functions for datatypes of kind `(*)`. None of them has a higher-order type, because interesting higher-order functions usually use their functional arguments on values belonging to their type variables. Seen from the perspective of parametric polymorphism, datatypes from kind `(* -> *)` upwards usually serve as a kind of structured container for element types inserted via type variables; this is when higher-order functions get powerful. Notable exceptions are functions not working immediately on data structures like `(.)`, `flip`, `($)`, and `until`. That is the reason why the most common higher-order functions `map`, `fold`, `filter`, and `scan` are not generalised for `Abs0`. Instead we have generalised typical functions which in some way are related to structure traversal without considering elements contained within the structure.

4.6.2 Functions for `Abs1` and `Abs2`

Much more interesting are functions for data structures of at least kind `(* -> *)`. These include the large set of omnipresent list-processing functions for which languages like Haskell are known. These and other functions can in general be categorised in terms of their 'data structure transformation' behaviour; by this we mean how the main argument data structures relate to result data structures seen in a 1-1 relationship. For our use we have roughly identified four categories for each of which we have constructed generalised representatives (see Figure 4.5):

- $(t\ a_1 \ldots a_n \rightarrow t\ b_1 \ldots b_m)$.
  Functions of this category *transform* the interior of a data structure and leave its overall shape mostly unchanged. Well-known examples include `map`, `zipWith`, and `unzip` which are all shown in their generic version in Figure 4.5. They all first decompose their argument, apply their non-generic counterparts to the components, and then reconstruct the value. Higher kinds are treated by applying additional arguments to additional components.

- $(t\ a_1 \ldots a_n \rightarrow [t\ a_1 \ldots a_n])$.
  These functions *select* parts of a data structure. The standard example `filter` has been generalised and is implemented and extended to higher kinds like the functions of the last category.

- $(t\ a_1 \ldots a_n \rightarrow b)$.
  Fold functions (or catamorphisms) reduce data structures and are represented by the generalised `foldr`. As the existence of a nullary constructor cannot be preassumed, a function `[a] -> b` has to be provided instead of the usual `b`. Two versions of `size` are presented, one defined in the usual way and one defined via `gfoldr1`. `gfoldr1` works by recursively folding the data structure tree. When lifted to higher kinds both of its argument functions have to be extended.

- $(b \rightarrow t\ a_1 \rightarrow a_n)$.
  To reduce is much easier than to *unfold*[71] , which is the basis for lazy constructive functions like `repeat` or `iterate`. Unfolding is difficult because a set of constructors can give rise to a wealth of possible unfoldings which would have to be generated fairly. Therefore we restrict ourselves to a pseudo-generic version which unfolds over a non-recursive and a recursive constructor given as arguments. (We do not use our `Abs` approach, which is in general not well suited for unfolding as it depends on a deconstruction prior to a construction.) As an example we have also shown a generic `repeat`.

Two standard selection functions have not been given in Figure 4.5, but are listed now. Generic version of `head` and `tail` are:

```
head1 :: Abs1 t => t a -> [a]
head1 = fst . sf . suco1

tail1 :: Abs1 t => t a -> [t a]
tail1 = snd . sf . suco1
```

```
-- Transform ---------------------------------------------------------
gmap1 :: Abs1 t => (a -> b) -> t a -> t b
gmap1 f t = rec (as', tas') where (_, ((as, tas), rec)) = suco1 t
                                  as'  = map f as
                                  tas' = map (gmap1 f) tas

gZipWith1 :: Abs1 t => (a -> b -> c) -> t a -> t b -> t c
gZipWith1 f t1 t2 = rec1 (cs, tcs)
  where (n1, ((as,tas),rec1)) = suco1 t1
        (n2, ((bs,tbs),rec2)) = suco1 t2
        cs                    = zipWith f as bs
        tcs                   = zipWith (gZipWith1 f) tas tbs

gunzip1 :: Abs1 t => t (a,b) -> (t a, t b)
gunzip1 t = (rec (as1, ta1), rec (as2, ta2))
  where (_, ((as,tas),rec)) = suco1 t
        (as1, as2)          = unzip as
        (ta1, ta2)          = unzip [gunzip1 t' | t' <- tas]
-- Select ------------------------------------------------------------
gfilter1 :: Abs1 t => (a -> Bool) -> t a -> [a]
gfilter1 f t = ls ++ rs
  where (_, ((as,tas), _)) = suco1 t
        ls                 = filter f as
        rs                 = concat [gfilter1 f ta | ta <- tas]
-- Fold --------------------------------------------------------------
gfoldr1 :: Abs1 t => ([a] -> [b] -> b) -> ([a] -> b) -> t a -> b
gfoldr1 f z t = if null tas then (z as) else (f as bs)
  where (_, ((as,tas), _)) = suco1 t
        bs                 = [gfoldr1 f z t' | t' <- tas]

size1, size1' :: Abs1 t => t a -> Int
size1 t = size_es + size_bs where (es, bs) = sf (suco1 t)
                                  size_bs  = sum (map size1 bs)
                                  size_es  = length es
size1' = gfoldr1 (\as bs -> length as + sum bs) (\as -> length as)
-- Unfold (pseudo-generic, without using Abs) ------------------------
gunfold1 :: ([a] -> ta) -> ([a] -> [ta] -> ta) ->
            (b -> Bool) -> (b -> a) -> (b -> b) -> b -> ta
gunfold1 nc rc p f g x | p x       = nc (repeat a')
                       | otherwise = rc (repeat a') (repeat ta')
  where a'  = f x
        ta' = gunfold1 nc rc p f g (g x)

grepeat1 :: ([a] -> t a) -> ([a] -> [t a] -> t a) -> a -> t a
grepeat1 nk rk = gunfold1 nk rk p f g where {p _ = False; f = id; g = id}
```

Figure 4.5: Sequential generic functions `(* -> *)`, based on `Abs1`

```
gmap2 :: Abs2 t => (a -> c) -> (b -> d) -> t a b -> t c d
gmap2 f g t = rec (as', bs', aas', abs', bas', bbs')
  where (_, ((as, bs, aas, abs, bas, bbs),rec)) = suco2 t
        as'  = map f as
        bs'  = map g bs
        aas' = map (gmap2 f f) aas
        abs' = map (gmap2 f g) abs
        bas' = map (gmap2 g f) bas
        bbs' = map (gmap2 g g) bbs

gZipWith2  :: Abs2 t =>
              (a -> c -> e) -> (b -> d -> f) -> t a b -> t c d -> t e f
gZipWith2 f g t1 t2 = rec1 (es,fs,tees,tefs,tfes,tffs)
  where (n1, ((as,bs,taas,tabs,tbas,tbbs),rec1)) = suco2 t1
        (n2, ((cs,ds,tccs,tcds,tdcs,tdds),rec2)) = suco2 t2
        es   = zipWith f as cs
        fs   = zipWith g bs ds
        tees = zipWith (gZipWith2 f f) taas tccs
        tefs = zipWith (gZipWith2 f g) tabs tcds
        tfes = zipWith (gZipWith2 g f) tbas tdcs
        tffs = zipWith (gZipWith2 g g) tbbs tdds

gfoldr2 :: Abs2 t =>
           ([a] -> [b] -> [c] -> c) -> ([a] -> [b] -> c) -> t a b -> c
gfoldr2 f z t | null tabs = z as bs
              | otherwise = f as bs cs
  where (_, ((as,bs,taas,tabs,tbas,tbbs),rec)) = suco2 t
        cs   = [gfoldr2 f z t' | t' <- tabs]
```

Figure 4.6: Sequential generic functions `(* -> * -> *)`, based on `Abs2`

Additionally Figure 4.6 shows selected functions of Figure 4.5 for kind `(* -> * -> *)`. They have been constructed according to the descriptions within the categories and are therefore not explained further. The amount of possible type variable sequences already shows that the taken approach cannot easily be lifted to larger flat kinds. On the other hand, datatypes with higher kinds are quite rare, with tuples being a common exception. If larger kinds become strictly necessary, one could easily turn away from covering types which flip their type parameters and save the huge effort of treating every possible sequence.

Until now we have only shown generic versions of the classical higher-order functions. These can of course be used to span a huge space of more specialised generic functions, as known from the standard Haskell prelude using their non-generic counterparts. For completeness, we will now give a couple of useful generic functions derived as specialised applications of `gfoldr1`:

```
and1, or1 :: Abs1 t => t Bool -> Bool
and1 = gfoldr1 (\as bs -> (and as) && (and bs)) (\as -> and as)
```

```
or1  = gfoldr1 (\as bs -> (or as)  || (or bs))  (\as -> or as)

any1, all1 :: Abs1 t => (a -> Bool) -> t a -> Bool
any1 p = or1  . gmap1 p
all1 p = and1 . gmap1 p

sum1, prod :: (Abs1 t, Num a) => t a -> a
sum1  = gfoldr1 (\as bs -> (sum as)  + (sum bs))  (\as -> sum as)
prod1 = gfoldr1 (\as bs -> (prod as) * (prod bs)) (\as -> prod as)

flatten1 :: Abs1 t => t a -> [a]
flatten1 = gfoldr1 (\as bs -> as ++ (concat bs)) id

elem1 :: (Abs1 t, Eq a) => a -> t a -> Bool
elem1 e = gfoldr1 (\as bs -> (elem e as) || (or bs)) (\as -> elem e as)

lookup1 :: (Abs1 t, Eq a) => a -> t (a,b) -> [b]
lookup1 e = gfoldr1 (\as bs -> [b | (a,b) <- as, a == e] ++ (concat bs))
                    (\as    -> [b | (a,b) <- as, a == e])
```

After having described our approach and having listed a couple of generic function, we will now discuss its characteristics.

4.7 Discussion

Now we will summarise the limitations of our approach. We also discuss objections which could be raised against it in comparison to other approaches.

- Within a datatype `(t a)`, our approach does not allow special access to values of other types than `a` or `(t a)` and is therefore not as complete as other approaches.

  As we aim at generic parallelism, our approach is not element-oriented but structure-oriented. Generic parallel functions aim primarily at processing and transforming large structures, not so much at quickly traversing an arbitrary structure to peek out a special element.

- Type contexts are introduced and may spoil the program code to some degree.

  The presence of type contexts cannot be avoided when using type classes. On the other hand, as an annotation they indicate use of generics within a function which helps to find these. The spine view introduced by Hinze, Löh, and Oliveira[101] avoids type classes, but at the cost of having to give explicit types on the value level.

- Datatypes of different flat kinds are not handled uniformly; instead a set of `Abs` classes has to be used.

  This is necessary as we

  - use explicit typing for the de- and reconstruction functions (instead of coding the types into expressions which would force us to implement our own typing)
  - type variable cannot be declared as sets

- Datatypes of higher-order kinds are not covered.

  This is true, but such datatypes are a rare case in parallel programming.

- Template Haskell has to be part of the Haskell implementation for running the preprocessor needed to automatically generate `Abs` instances.

  Template Haskell has become a standard part of the Glasgow Haskell Compiler is not likely to be dropped in the future. In the contrary, Template Haskell meta-programming can be expected to gather even higher interest.

- The approach provides only 'fake' genericity, as functions are not really generic but only for the datatypes (for which instances have been generated) of the corresponding program where they are used.

  This is true, but the difference is not observable. Other approaches also use this mechanism.

- The approach is quite verbose as explicit functions for de- and reconstructing data constructors are given. These have to be called explicitly and are visible in the program.

  Data constructors need to be handled somehow, and we do it in a recursive traversal. Usually calls to the `suco` functions do not appear in ordinary functions, but are hidden in higher-order functions which are imported as a separate module.

- The permanent value de- and reconstruction is expensive and decreases program performance.

  This is true. As an example, a traditional `map` is twice as fast as `gmap` applied to lists. This is due to the overhead caused by the dictionary lookup of the class mechanism and the extra calls to `suco`. But for our application of parallel programming, this is not as grave as one may think at first as:

  - Generic parallel functions are normally used in the large and not in the small. This means, that data structures are usually not deconstructed to the smallest element. Therefore, the overhead caused by genericity is not too big.
  - Generic functions are also a tool during software development. At performance-critical positions they can easily be specialised and replaced by faster versions. We consider it a rare case that a truly generic function is irreplacibly needed at a central position in a program.

- Due to its restrictive typing, the approach cannot handle non-uniform datatypes like

```
data Seq a = Nil | Cons a (Seq (a,a))
```

This is true. However, non-uniform datatypes can be reformulated to regain uniformity. The pair nesting can be defined implicitly by an integer depth parameter:

```
data Seq' a = Nil' | Cons' Int [a] (Seq' a)
```

Alternatively, one can always pull the pairwise nesting out of the recursive `Seq` and into a newly defined element datatype (like shown in Section 10.1.1 of Okasaki[161]):

```
data Seq'' a = Nil'' | Cons'' (Elem a) (Seq'' a)
data Elem  a = E a   | P (Elem a) (Elem a)
```

Unfortunately, this redefinition suffers from introducing additional data constructors and from giving up a tight typing which allowed the type system to catch many errors. The programmer has to ensure the correct nesting of elements.

- As flipped type elements are handled, the types of the `suco` functions will grow very fast.

  This is true, but types with more than three type variables are very rare in practice. On the other, flipped type variables are also rare; if necessary, the approach can most easily be changed to skip these.

- The approach is able to deconstruct and consume values, but not to build values.

  We have shown a limited way to building values by giving a reduced `unfold` definition. However, the approach has been designed for supporting parallelism where controlled deconstruction is much more important than value building.

5. Controlling Demand in Eden

Vertrauen ist gut, Kontrolle ist besser.

German saying

5.1 Motivation

There is one central aspect in lazy functional programming which can cause grave unexpected behaviours during program execution but is sometimes greatly underestimated: The *way evaluation progresses* influences the time and space needs of a program greatly. In lazy (or non-strict) languages the reduction order is clearly defined: Out of all possible reductions within a reducible expression the outermost reduction is done next. The *demand* for evaluation propagates through a program, which is a set of function declarations. In parallel functional languages like Eden this *demand flow* is closely related to parallelism, as demand is responsible for process creation. Eden itself is a language which deliberately puts full parallelism control into the hands of the programmer; besides a very carefully implemented parallel algorithm it is therefore equally important for the programmer to control demand flow which is responsible for the correct progress of the algorithm. In parallel functional programming, demand and parallelism cannot be separated.

In Section 5.2 we will first show how a functional program is executed and how reduction progresses. Section 5.3 then points out why demand has to be controlled in some situations both in sequential and especially in parallel programming. Having acknowledged the need for demand control we turn in Section 5.4 our attention to which means for demand control are present in Haskell. In Section 5.5 we focus on demand related to data structures. Using generic programming we show methods to flexibly evaluate data structures to different degrees. Finally in Section 5.6 we identify effects, to which not only data structure evaluation functions but also process creation and others belong. Besides getting a more structured program in the sense of aspect-orientation, the separation and identification of effect groups are the key to true parallelism. In the end we propose three measures for *demand management*, which together form a tool for extensive demand control in parallel functional programs.

Parts of the chapter are based on two of our papers[118, 174] .

5.2 Sequential Evaluation of Functional Expressions

5.2.1 How Expressions are Reduced

In Haskell a functional program consists of a set of function declarations. A *target expression* containing applications of functions defined in this set is evaluated to finally yield a result value. Execution of a functional program corresponds to the continuous evaluation of reducible subexpressions within the target expression and the replacement of the subexpression by its value. Expressions can contain many reducible subexpressions which can also be nested. This means, that expressions can be reduced to different degrees:

5.1 Definition (Reduction Depths)
An expression e *is in*

- normal form (NF)*, if it contains no reducible subexpression (also called* redex *for short).*
- weak head normal form (WHNF)*, if it is either*[201] *:*
  - *a value of a base type*
  - *a data constructor* `(C e1 ... ek)` *where* C *has at least arity* k
  - *a lambda abstraction* `\x -> e`
  - *a partial function application* `(f e1 ... el)` *where* f *is* n*-ary and* $l < n$

If more than one reducible subexpression exists, a choice has to be made which one to reduce next. Two different *reduction orders* are common:

5.2 Definition (Reduction Orders)

- *Within the* strict *reduction order, always the* leftmost innermost *redex is reduced next (also known as* call-by-value*).*
- *Within the* non-strict *reduction order, always the* leftmost outermost *redex is reduced next (also known as* call-by-name *or* demand-driven*).*

Leftmost-innermost reduction means, that regarding a function application at first all function arguments are evaluated before evaluation of the function body starts (also called *eager evaluation*). Eventually the function body will be reduced to normal form. This means, that a non-terminating argument expression will unfortunately cause the non-termination of the whole function application. This is even more unfortunate if the diverging argument is not needed to compute the function's result. For example, given a function `f` a call with a non-terminating first argument causes the whole application to diverge:

`f x y = y`, $\quad$ `f` $\bot\ 2 \rightarrow \bot$

Advantages of strict evaluation are its ease of implementation and the obvious way to exploit parallelism by simultaneous evaluation of argument expressions.

On the other hand, leftmost-outermost reduction means, that regarding a function application the function's body will be evaluated first. The evaluation of the arguments is deferred until they are needed to produce the function's result; if an argument is never needed, it will remain unevaluated. This is especially advantageous, because non-terminating arguments which are not needed will not cause the whole function to diverge. The danger of evaluating the same expression more than once, like in a simple multiplication

```
(nth_prime 42) * (nth_prime 42)
```

is averted by the *sharing* mechanism, by which identical subexpressions are assigned the same memory location. After evaluating the expression for the first time, it will be replaced by its value. Therefore subsequent evaluations of that expression will immediately get the result value. The combination of the non-strict reduction order and the sharing mechanism is also called *lazy evaluation*. It enables circular definitions, formally infinite data structures, and allows for runtime savings by referencing results which have not yet been produced. The main disadvantage is a more complex implementation as suspended evaluations have to be administrated.

5.2.2 How Reduction Progresses

A special function called `main` is singled out of the set of function declarations within a program. Via the `IO` monad `main` is the root of every safe I/O operation within the program and therefore the connection to the console and the file system. The whole program starts by evaluating `main`, which most often consists of an I/O operation writing the result value of a function call representing the actual algorithmic calculations (called *target expression* for now) to the console or a file, for example:

```
main :: IO ()
main = putStr (show (f x1 ... xn))      -- write as string to console
                    \_____________/
                    Target expression
```

Other possible I/O operations are left aside for now. As of course unevaluated function calls cannot be written to any media, the result of the function call has to be completely evaluated and free of any unreduced function calls. This means, that it is either a value belonging to one of Haskell's base types (like `Char`, `Bool`, `Integer`) or it is a (probably nested) data constructor. Besides, base type values can also be seen as nullary data constructors.

This means, that in the non-strict language Haskell the `main` function is the initial source of all *demand* for evaluation. The I/O functions contained within `main` exert constant demand for evaluating expressions to yield result values. In general, an expression representing a result data structure is demanded from outside until the demand is satisfied, which means that the result data structure has been established to a sufficient degree. Expression evaluation works stepwise:

the leftmost-outermost redex is reduced until weak head normal form has been reached. If a satisfying evaluation degree has been reached the evaluation will stop. Otherwise the next leftmost outermost redex is evaluated. In total, there are only three common natural sources for demand:

- *I/O operations* (like output calls in `main`) exert the demand necessary to transform the target expression into a value suitable for output. If that demand is satisfied, no other demand for further evaluation can exist (in non-parallel languages).
- *Strict basic functions* like arithmetic and comparison operators demand all of their arguments to yield a result.
- *Case expressions* like

  ```
  case e of { pat1 match1 ; ... ; patn matchn }
  ```

  employ pattern-matching via `pat1` to `patn` to distinguish the available alternatives. *Irrefutable* patterns like variables, wildcards, `@`-patterns, or lazy matching via `~` will always match any argument and leave that argument unchanged. In contrast there are also *refutable* patterns like explicit base type values or data constructors (including tuples). To be able to decide which alternative matches, the expression `e` upon entering `case` has to be evaluated to such a degree that allows the decision to be made. Thus `case` is another source of demand. Demand is also generated by guards[164] , direct pattern-matching in function definitions or `let` expressions, and conditional expressions; but all these are in the end `case` expressions.

Based on the importance of demand control, some more special constructs for demand steering have been added to Haskell. These are described in Section 5.4 in detail.

If these sources of demand are used in a function definition, the function exhibits a termination behaviour depending on its arguments. This can happen if strictly needed arguments are non-terminating. This is formally captured by the strictness property:

5.3 Definition (Strictness)
There are two notions of strictness:

- *A function* $f : t_1 \to \ldots \to t_n \to t$ *is called* strict in its i-th argument *(with* $i \in \{1, \ldots, n\}$*) if*

  $\forall x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ *with* $x_j :: t_j$*:* $\quad f\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n$ *evaluates to* $\perp$

  ($\perp$ *represents a non-terminating evaluation.)*
- *The same function is called* strict*, if it is strict in every argument.*

The following example demonstrates how evaluation of an expression proceeds and where demand takes effect. It is a very basic example which serves only for clarification.

3 Example (Non-strict reduction)
We define a binary search tree and a function `ins` for inserting elements into such a tree. `ins` contains refutable patterns and a `case` expression.

```
data BST a = E | Node (BST a) a (BST a)             -- (E)mpty or Node

ins :: Ord a => a -> BST a -> BST a                 -- insert element
ins x E            = Node E x E
ins x (Node l y r) = case x <= y of
                       True  -> Node (ins x l) y r
                       False -> Node l y (ins x r)

ex = Node (Node E 2 E) (105 `mod` 10) (Node E 6 E)  -- example BST
```

Below the insertion of an element into an example tree is shown. On the left the expression is reduced step by step; dots mark not yet demanded subexpressions. On the right the corresponding (assumed) console output is shown as it evolves. Note how `case` forces the evaluation of `(105 `mod` 10)` in line 3. Note also how console output forces evaluation to look up an `Empty` node in line 7.

```
1     ins (2+2) ex                                 |
2  -> ins (2+2) (Node . . .)                       |
3  -> ins 4 (Node . (105 `mod` 10) .)              |
4  -> ins 4 (Node . 5 .)                           |
5  -> Node (ins 4 .) 5 .                           |  Node
6  -> Node (ins 4 (Node . 2 .)) 5 .                |
7  -> Node (Node . 2 (ins 4 .)) 5 .                |  (Node
8  -> Node (Node E 2 (ins 4 .)) 5 .                |  E 2
9  -> Node (Node E 2 (ins 4 E)) 5 .                |
10 -> Node (Node E 2 (Node E 4 E)) 5 .             |  (Node E 4 E)) 5
11 -> Node (Node E 2 (Node E 4 E)) 5 (Node E 6 E)  |  (Node E 6 E)
```

The example has shown a typical evaluation sequence. Unfortunately, there are situations which can cause evaluation to proceed in an unfavourable way not intended. These will be discussed in the next section; before doing so, we will first shortly describe additional reduction degrees.

5.2.3 Data Structure Reduction Degrees

The strictness of a function, which we have described in the definition above, is an interesting property which can be used in many ways. *Strictness analysis*[188] is used to gather strictness information. To do that, the actual value domain of a data structure is mapped to an abstract domain whose values represent the needed strictness information. Until now we have been interested only in the fact whether a value is completely unavailable (described by $\perp$) or not. This is sufficient for flat[57] types like `Int`, `Bool`, or `Char`. But with more complex self-defined non-flat

data structures more complex strictness statements can be made[212] , as these structures allow for more complex combinations of definedness (as has been hinted by Def. 5.1).

If we take the list structure on characters as an example, it is immediately clear, that it generates an infinite abstract domain:

```
undefined
undefined : undefined
def : undefined
def : []
undefined : undefined : undefined
def : undefined : undefined
...
```

To be manageable, domain sizes have to reduced by combining similar cases. Figure 5.1 shows a common reduction to a four-point domain. Besides being fully

Example	Description	Burn[35]	Wadler[212]
`undefined`	Completely undefined	E0	$\bot$
`undefined:undefined`	WHNF	E1	∞
`'a':undefined:'c':[]`	Full list spine	E2	$\bot_\in$
`'a':'b':'c':[]`	Full spine and all elements	E3	$\top_\in$

Figure 5.1: Reduction degrees of `[a]`

defined and fully undefined the two middle cases are interesting. While the second case only touches the first data constructor without descending further, the third case completely traverses the spine of the data structure (every recursive limb of type `[a]`) but does not touch any value of type `a`. These cases have become very common and will be used in a later section of this chapter for a generalised reduction function.

5.3 The Need for Demand Control

The picture drawn so far is tempting: Whenever strict reduction terminates, non-strict reduction will also terminate, except that unnecessary evaluations are omitted. However, it is not all roses. In this section we show for both the sequential and the parallel Eden setting why it is mandatory to take care of demand control.

5.3.1 Sequential Setting

In a sequential setting the amount of demand is important regarding memory usage. Usually functions and data structures are stored within a heap structure and most often form by and large a tree with a couple of interconnections and cycles. Nodes of that graph are called *closures*, which contain values or unevaluated expressions. If heap space is tight, a *garbage collection* is triggered which traverses

the graph starting from the result expression to evacuate everything still needed to calculate the result. Intermediate or cut-off data structures and expressions are not referenced by the graph, are therefore not evacuated, and finally disposed as garbage. The following describes unfavourable situations in which heap space shortages are related to unintended demand behaviour:

Not enough demand.
Too much laziness can do harm, as the following example shows:

---

4 Example (Stack overflow)
Imagine one wants to sum up all numbers within a list. A `foldr` seems to be appropriate, but the strictness of `(+)` demands that the second argument must be known. Therefore a large nested arithmetic expression is built which results in heavy stack load.

```
   foldr (+) 0 [1..100000]
-> 1+(foldr (+) 0 [2..100000])
-> 1+(2+(foldr (+) 0 [3..100000]))
-> 1+(2+(3+(foldr (+) 0 [4..100000])))
-> ...
-> stack overflow!
```

`foldl` does not remedy that: It builds up a nested arithmetic expression which can only be evaluated when the end of the list has been reached. The leftmost-outermost redex is always `foldl`.

```
   foldl (+) 0 [1..100000]
-> foldl (+) (0+1) [2..100000]
-> foldl (+) ((0+1)+2) [3..100000]
-> foldl (+) (((0+1)+2)+3) [4..100000]
-> ...
-> stack overflow!
```

These overflows are also called *space leaks*, because either heap or stack are burdened with structures which are much bigger than expected. This will cause the stack to overflow and the heap to trigger expensive garbage collections or also to overflow. The reason is typically not enough demand (or excessive laziness) at the right place and at the right time. Eventually these expressions are evaluated and (most often) deflated to comparatively small values while the large expressions which have created them are garbage collected later. The `foldl` example above can be repaired by adding just enough strictness to the `foldl` definition to demand the immediate addition of values for every list element processed. The means to do that together with the solution are presented in Section 5.4.

Not enough / too much demand.
Closures can be quite big, as they are also equipped with infotables and (depending on the compiler) are piggybacked with additional parameters for profiling, parallelism, etc. This raises the question whether it is cheaper to store an unevaluated expression or to evaluate the expression and to store the value. If the value takes

up less space and has to be evaluated anyway, then it should be evaluated early by placing additional demand. This is especially important in the context of lazy evaluation, where demand may walk through many expressions leaving an inflated pile of large, partially evaluated WHNF expressions behind. These often use intermediate data structures which are not needed otherwise; but being referenced by these incompletely evaluated expressions keeps them from being exposed to garbage collection. The space used by these intermediate structures could be deallocated if the expressions would have been fully evaluated. On the other hand, if the value generated by an expression takes up more space than its creator, it is better to evaluate not too much:

Too much demand.
Too much demand can also cause heap space shortages. Mostly this is caused by defining and evaluating *constant applicative forms* or *CAFs* for short. These are simply top-level constants which one easily tends to define, like:

```
nats   = [1,2..]               -- list all natural numbers
matrix = buildMat 1000 1000    -- build 1000 x 1000 matrix
```

Unfortunately, these can permanently occupy much heap space as heap referenced by CAFs is never garbage collected. Although modern Haskell compilers try to avoid creating CAFs, a CAF intentionally created by the programmer is usually not eliminated. Therefore special care has to be taken when evaluating CAFs, as every newly created part will persist until the program ends.

To be able to influence evaluation is helpful in a sequential setting, but becomes vital in a parallel setting as we will show now.

5.3.2 Parallel Setting

Because of their independence the parallel evaluation of function arguments upon entering a function is an obvious occasion for the easy introduction of parallelism. This is the reason why strict functional languages are often favoured as a basis for parallel extensions. But if one wants to introduce parallelism and retain the advantages of non-strict reduction (like the ability to define circular structures), then the evolution and control of demand becomes a matter of high importance.

As shown before, Eden's language constructs for parallelism are the *process abstraction* and the *process application*. Denotationally these correspond to the well-known lambda abstraction and lambda application. However, operationally they differ because in Eden a process for evaluating the application is generated on a network node. For the following example, let us assume the following conservative Eden semantics (in contrast to its real semantics described in Section 2.5) by which non-strict reduction is adopted for Eden's language constructs:

- A process is not created before its result is really needed.
- Results are evaluated to WHNF. Then reduction stops and is not resumed before new demand for larger parts of the results has been exerted.
- Results are transmitted only on demand.

The following example shows an Eden program for calculating Hamming numbers [61] and Figure 5.2 shows how it is executed when the assumptions shown above are made. The calculation of the Hamming numbers is done via a set of interacting streams using circular references to the result list. This is possible because reduction is carried out non-strictly.

---

5 Example (Hamming numbers)
For a given non-empty set of primes $\{p_0, \ldots, p_{m-1}\}$ and a given $n \in \mathbb{N}$ enumerate all elements of

$$\{z \mid z = p_0^{k_1} * \ldots * p_{m-1}^{k_{m-1}}, k_i \in \mathbb{N}_0, z \leq n\}$$

without multiples and in ascending order. For the set of primes $\{2, 3, 5\}$, for which the first 20 Hamming numbers are

$$\{1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36\},$$

this is done by generating a `multStream` process for each prime. Each `multStream` multiplies every element of the incoming stream with a previously assigned prime number. The input stream is simply the result list `hamming`, which ensures that every process will also compute multiples of the results of other processes. All three streams are ordered (including evaluation of multiple elements) by a `mergeThree` process. To illustrate demand progression, those subexpressions for which there is immediate demand are boxed.

```
1  multStream :: (Num a, Trans a) => a -> Process [a] [a]
2  multStream n = process (map (*n))
3
4  mergeThree :: (Ord a, Trans a) => Process ([a],[a],[a]) [a]
5  mergeThree = process (\(xs,ys,zs) -> sm xs (sm ys zs))
6    where sm []     ys     = ys
7          sm xs     []     = xs
8          sm (x:xs) (y:ys) | x <  y = x : sm xs     (y:ys)
9                           | x == y =     sm (x:xs) ys
10                          | x >  y = y : sm (x:xs) ys
11
12 hamming :: [Int]
13 hamming = 1 : mergeThree # ((multStream 2) # hamming,
14                             (multStream 3) # hamming,
15                             (multStream 5) # hamming)
```

The reduction sequence clearly shows, that it is not sufficient to just *generate processes*. Parallelism emerges and time savings are obtained if processes coexist whose active phases *overlap*. In our example, the demand-driven evaluation does not generate such overlaps. At every point in time, only one process is actively evaluating. There is only one flow of demand which is never split to obtain results in parallel. All non-active processes are waiting to regain demand. This situation of *distributed sequentiality*[118] shows the vital need for demand control to enable true parallelism. But how much more demand is needed? In a sense non-strict reduction and parallelism are two conflicting aims: On the one hand, we would like to defer expression evaluation to the latest possible point in time so that we

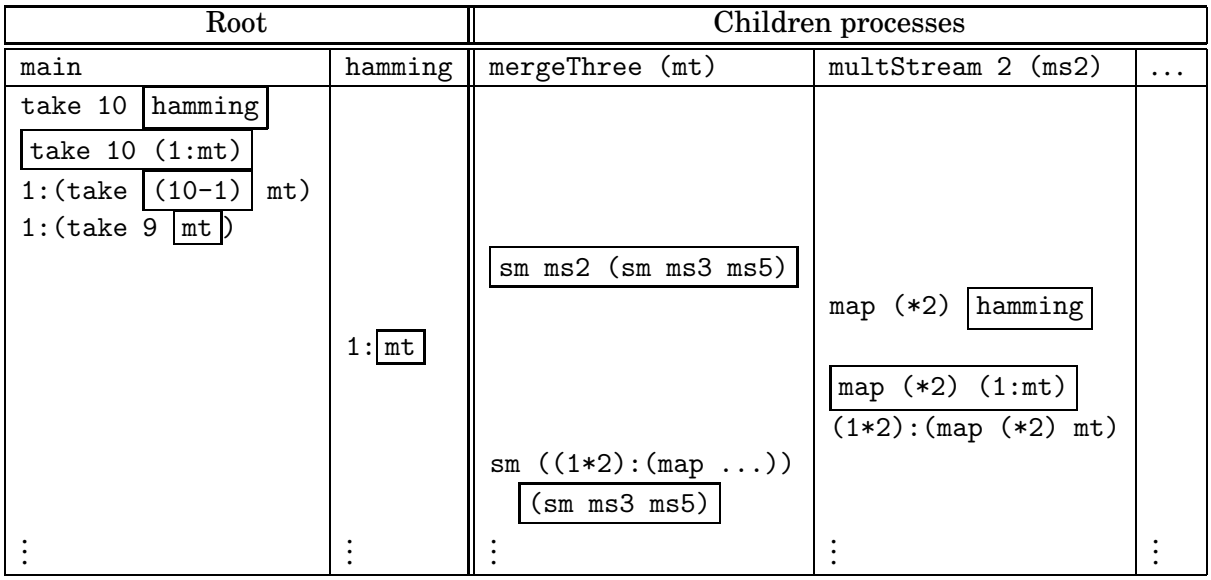


Figure 5.2: Lazy reduction of `hamming` demonstrating distributed sequentiality

maybe in the end could even let it remain unevaluated. On the other hand, the earliest possible evaluation supports parallelism by generating processes as early as possible. Therefore we have to make decisions on

- *how* demand can be controlled
- *how much* demand is exerted and finally
- *when* and
- *where* it should be enforced.

In the next section we will answer the first question. The remaining questions are answered in subsequent sections.

5.4 Means for Demand Control

The paradigm of lazy evaluation is usually seen as being quite optimal in the sense that no unnecessary evaluation is done. As we have seen before, this is true but this can also be carried too far by postponing *too many* evaluations to a later point in time. Then the problem changes from avoiding unneeded computations to avoiding excessive laziness. The needed demand control to solve the problem is often considered dirty or non-standard, breaking the smooth non-strict evaluation order and spoiling the otherwise pure flow of demand. On the other hand, its usefulness has been widely acknowledged as there are surprisingly many approaches and possibilities for controlling demand in Haskell 98[164] . Some research has also been done on that topic which we discuss in the chapter about related work. As Eden is based on Haskell we will now give a systematic overview about ways of controlling demand in Haskell; these will then also be applicable for Eden.

5.4.1 Without Operators

We will start by looking at purely functional ways of demand control. Neither operators nor classes are used.

Implicit Eden Demand. Eden introduces additional demand into Haskell. If a process is created by

$$(res_1, \ldots, res_m) \;=\; p \;\#\; (arg_1, \ldots, arg_n)$$

then Eden will force evaluation of $res_1, \ldots, res_n$ without caring about the real demand for these expressions. Furthermore, as already discussed in our master thesis[171] , the demand for an element out_i of a result tuple created by a process application

$$(\underbrace{out_1,\ \ldots,\ out_{i-1},}_{\Rightarrow \textit{ new demand}}\ \overbrace{out_i}^{\textit{initial demand}},\ \underbrace{out_{i+1},\ \ldots,\ out_n}_{\Rightarrow \textit{ new demand}}) \;=\; pabs\ par_1\ \ldots\ par_k \;\#\; \underbrace{(in_1,\ \ldots,\ in_m)}_{\Rightarrow \textit{new demand}}$$

has two consequences: Firstly, process creation implies the speculative evaluation of $out_1, \ldots, out_{i-1}, out_{i+1}, \ldots, out_n$ which may be unneeded. Secondly, new demand is placed on the argument expressions $in_1, \ldots, in_m$, which, in the case of streams, could cause a continuous evaluation and value transmission sequence. On the receiving side, this is accompanied by an increasing heap consumption while the arguments are being referenced.

Deep Patterns. In Haskell a function is declared via a sequence of alternatives which cover different argument constellations. If the function is used, a fitting alternative (more than one may fit) is found by traversing the alternatives sequentially. To decide whether an alternative matches the given arguments are evaluated to such a degree that the decision can be made. Pattern-matching, i.e. `case`, can therefore be used to trigger the evaluation of a data structure to a fixed depth by giving nested data constructors as patterns:

```
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)

f (Node (Leaf x) y (Leaf z)) = ...
```

As patterns may overlap, Haskell follows a first-fit policy. Therefore the more special pattern alternatives usually precede the more general ones which are finally closed by a catch-all alternative using the wildcard operator. Unfortunately, patterns have to be statically fixed. This prevents flexible demand control.

Nested Case. As `case e of ...` exerts demand on the scrutinee `e` to allow choosing the first matching alternative, a series of nested `case` expressions can be used to demand a series of expressions:

```
case e1 of
  r1 -> case e2 of
          r2 -> ...
```

This simple construction is of course in danger of being optimised away, as the `case` results are maybe unused. Also other compiler-internal transformations can change the intended behaviour: case floating, as described by Santos[185] , may change the evaluation order of `e1` and `e2`. The `seq` operator, described in the next subsection, preserves that order.

Data Dependencies. Demand steering is also possible by enforcing an evaluation order via data dependencies. This is how *state transformers*[132] introduce *sequentiality* into non-strict languages. A state transformer takes a state and yields a new state together with a result which is fed into the next state transformer. A sequence is built via the `>>=` operator, which connects two state transformers. The state resulting from executing the first one is fed as a starting state into the second one:

```
data ST s a = ST (s -> (a, s))

instance Monad (ST s) where
  (ST m) >>= f  = ST (\s -> let (v, s') = m s
                                ST m'   = f v
                            in m' s')
  return v      = ST (\s -> (v, s))
```

This approach is quite costly and complex, as for systematic demand control large parts of a program would be dominated by the state transformer monad.

Continuation Passing Style. To enforce a certain evaluation order, also *continuation passing style (CPS)*[10] can be used. Imagine a function `f` defined as follows:

```
f x y = g (h x y)
  where h a b = result
```

Inside `f`, a call to `h` is wrapped into a call to `g`. Due to lazy evaluation, `h` will at first not be computed. This can be changed by turning the expression inside out: To expose the call to `h` to evaluation, it now forms the result of `f`. A new argument has been added to `h`, which is called a *continuation*. It represents the functionality of `f` and is passed as a parameter to `h` which applies its result to it.

```
f x y = h x y (\res -> g res)
  where h a b cont = cont result
```

This way `h` is reduced before entering `g` and a certain sequence of reduction is enforced. For recursive functions `h`, passing along the continuation gets more complicated. And for even more complex functions, the redesign of the whole function may be necessary. In general, CPS works but can heavily complicate a program.

The last two approaches are only useful for controlling the reduction *order*. This way no additional demand is created, therefore means like pattern-matching have

to be used in combination with these. Due to their difficult applicability, additional strictness operators have been introduced.

5.4.2 With Operators

Special strictness operators and annotations ease demand control as they can easily be used more abstractly.

The `seq` combinator. In the context of another parallel Haskell dialect, *Glasgow Parallel Haskell (GpH)*[205] , the need for demand control had also arisen. Beneath an operator for introducing parallelism called `par` a second operator for demand control was introduced:

```
seq :: a -> b -> b
```

The semantics of `seq` is simple but powerful: The first element is (as a side-effect) evaluated to WHNF and the second argument is given back as a result:

$$\texttt{seq}\ x\ y = \begin{cases} \bot, & \text{if } x = \bot \\ y, & \text{otherwise} \end{cases}$$

`seq` cannot be defined directly in Haskell. Instead, in the Glasgow Haskell Compiler GHC, it is defined as a unique identifier whose type and implementation is supplied internally. Its implementation corresponds in essence to a case application like

```
seq x y = case x of _ -> y
```

which is protected from compiler transformations. Using `seq`, another function `pseq` is defined:

```
pseq :: a -> b -> b
pseq x y = x `seq` lazy y
```

Without `lazy`, the strictness analyser would find out that `pseq` is strict in `y`; then the compiler could via transformations evaluate `y` before evaluating `x`. As this is not desired, a function `lazy` is defined as a unique identifier which prevents that:

```
lazy :: a -> a
lazy x = x
```

`lazy` behaves like `id` but its strictness is not defined by the strictness analyser; instead, the strictness information is overridden internally and set to non-strict. This way the evaluation order is preserved.

Using infix notation one can then control demand via a sequence of `seq` combinators. If demand control is about to be introduced into a function definition

```
f x y z = h <exp_x> <exp_y> <exp_z>
```

one most often gives names to the subexpressions one wants to control and then exerts demand on each of them:

```
f x y z = let expx = <exp_x>
              expy = <exp_y>
              expz = <exp_z>
          in expx `seq` expy `seq` expz `seq`
             h expx expy expz
```

Strict apply. Starting from `seq` many other operators can be derived. A useful one defined in the Haskell standard prelude is a strict apply operator

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

which is equal to normal function application except that the function argument is evaluated to WHNF before being applied to the function.

Strict fields. Fields within data constructors can be labelled with a strictness annotation. While the arguments of `C1` are unlabelled both arguments of `C2` are given a strictness flag which causes them to be evaluated when the data constructor is defined:

```
data T a b = C1  a  b
           | C2 !a !b
```

Guard tricks. The `seq` operator shown above can be used to trigger evaluation upon entering a function before anything else has happened. As Haskell uses a first-fit pattern-matching semantics, we add an alternative (standing before all other alternatives of the function) which always fails in a guard. Before failing, effects can be carried out. A function `f` can be modified in the following way:

```
                          f x y | effects `seq` False = undefined
f x y = result     →            where effects = ...
                          f x y = result
```

When looking for a fitting alternative Haskell's pattern-matching will first check the effect alternative, execute the effect, and then fail. Afterwards, the original alternatives will be checked as before. This trick is due to Oleg Kiselyov and has been described in the Haskell mailing list.

Another older trick to force evaluation is to add an equality check via a guard. Unfortunately, newer compilers optimise these checks away and make the trick useless. Additionally, it works only on types for which instances of `Eq` are defined. Also, one cannot rely on correctly defined instances which really execute the needed evaluation.

```
eval :: Eq a => a -> ()
eval x | x == x = ()
```

Equipped with these mechanisms, we can now solve the space leak problem shown in Section 5.3:

6 Example (Strict foldl)
The problem we encountered when summing up a list of numbers via a fold was a growing arithmetic expression which can only collapse when the fold has finished. Before that usually a heap or stack exhaustion happened. Now we can counter this via strictly evaluating the newly built expression within each folding step:

```
strictFoldl :: (a -> b -> a) -> a -> [b] -> a
strictFoldl (*) z []     = z
strictFoldl (*) z (x:xs) = ((strictFoldl (*)) $! (z*x)) xs
```

Now one gets the following reduction sequence (which runs in constant space) when summing up a list of numbers:

```
   strictFoldl (+) 0 [1..1000000]
-> ((strictFoldl (+)) $! (0+1)) [2..1000000]
-> strictFoldl (+) 1 [2..1000000]
-> ((strictFoldl (+)) $! (1+2)) [3..1000000]
-> strictFoldl (+) 3 [3..1000000]
-> ...
```

These mechanisms are sufficient for dealing with data structures which are small or of low complexity. More complex data structures however require a more powerful approach to allow for comfortable demand control.

5.4.3 With Operators and Overloading

The methods shown in the last section can ensure that an evaluated expression is at least in WHNF. Often this is sufficient, but on other occasions some more control is needed: One could for example want to evaluate an expression to NF, which, given only the above methods, is a tedious task. A more systematic approach is to use type classes to overload a common evaluation function. This function is then declared for most (or all) data structures by giving a corresponding instance of that type class.

DeepSeq. The *DeepSeq* class communicated by Dean Herington via the Haskell mailing list extends `seq` and `($!)` to `deepSeq` and `($!!)`. In contrast to `seq`, `deepSeq` descends into a data structure (evaluates its spine) and evaluates its leafs to WHNF. Parts of the class are provided in Figure 5.3. The missing parts are straightforward to define.

Figure 5.4 shows the same functionality while avoiding the use of type classes. Instead overloading is modelled by adding explicit type information to expressions via the `Typed` data structure. This technique has been propagated by Cheney and Hinze [39] and then been used by Hinze et al.[101] for the definition of generic functions. Types themselves are represented by a generalised algebraic data type [190] . This approach shifts the otherwise unreachable

```
class DeepSeq a where
  deepSeq :: a -> b -> b
  deepSeq = seq

infixr 0 `deepSeq`, $!!

($!!) :: DeepSeq a => (a -> b) -> a -> b
f $!! x = x `deepSeq` f x

instance DeepSeq ()                        -- Base type instances
instance DeepSeq Int
...

instance DeepSeq a => DeepSeq [a] where   -- Structured type instances
  deepSeq []     y = y
  deepSeq (x:xs) y = deepSeq x $ deepSeq xs y
...
```

Figure 5.3: The `deepSeq` type class

```
data Type :: * -> * where
  Unit :: Type ()
  Char :: Type Char
  Int  :: Type Int
  Pair :: Type a -> Type b -> Type (a,b)
  List :: Type a -> Type [a]

data Typed a = HasType {val :: a, typ :: Type a}

deepSeq :: Typed a -> b -> b
deepSeq (HasType v  Unit)       e = v `seq` e
deepSeq (HasType v  Char)       e = v `seq` e
deepSeq (HasType v  Int)        e = v `seq` e
deepSeq (HasType p  (Pair a b)) e = case p of
                                      (v1,v2) ->
                                        deepSeq (HasType v1 a) $
                                        deepSeq (HasType v2 b) e
deepSeq (HasType xs (List a))   e =
  case xs of
    []     -> e
    (y:ys) -> deepSeq (HasType y  a) $
              deepSeq (HasType ys (List a)) e
```

Figure 5.4: `deepSeq` via explicit type annotations and GADTs

type information to the language level where it can be used to assign different implementations to each type and gain an overloaded function. Within the `deepSeq` function this overloading shows up as every alternative handles a separate type case and assigns it its special implementation.

Strategies. *Evaluation strategies*[207] have been introduced into Glasgow Parallel Haskell as a method to structure complex uses of the `seq` and `par` combinators. Strategies are simply functions which have an effect on an argument value (see Figure 5.5) and yield unit as a result. Strategies are applied via the `using` operator. For sequential demand control a class `NFData` has been introduced which is equivalent to the `DeepSeq` class. Strategies get interesting when parallelism via the `par` combinator is introduced; then strategies combine both sequential and parallel control within a single tool.

```
type Strategy a = a -> ()

using :: a -> Strategy a -> a
using e st = st e `seq` e

class NFData a where
  rnf :: Strategy a
  rnf = rwhnf

instance NFData a => NFData [a] where
  rnf []     = ()
  rnf (x:xs) = rnf x `seq` rnf xs
```

Figure 5.5: Evaluation strategies, sequentially

Finally, there are many means for demand control. In the next subsection we will use these to provide more comfortable ways of demand control.

5.5 Data-Oriented Demand Steering

As functional programming is about values, we will start with looking for additional and more powerful demand steering mechanisms for data structures. These can be evaluated to different degrees and we will therefore focus on constructing functions for doing that in a comfortable way. Lastly we will discuss explicit data structure annotations; using these, a data structure can carry information about itself around during evaluation.

5.5.1 Spine Evaluation, Generically

In Figure 5.1 we have already discussed list spines. Now we will turn to a more general definition.

5.4 Definition (Spine)
Let v be a value of a data structure $T\ a_1 \ldots a_n$

$$\begin{array}{rcl} data\ T\ a_1 \ldots a_n & = & D_1\ t_{11} ... t_{1m_1} \\ & | & D_2\ t_{21} ... t_{2m_2} \\ & & \vdots \\ & | & D_k\ t_{k1} ... t_{km_k} \end{array}$$

where a_i are type variables, D_i are data constructors, and t_{ij} are types like external datatypes, type variables out of $a1, \ldots, a_n$, or the (recursive) data structure $Ta_1 \ldots a_n$ itself. The spine *of the value v is then the tree traversal of v such that only those limbs are visited which are of type $Ta_1 \ldots a_n$. All others are left unvisited.*

For example, a function for traversing the spine of a list could be written as:

```
spine_list :: [a] -> ()
spine_list []     = ()
spine_list (_:xs) = spine_list xs
```

In the second alternative, only the recursive branch of type `[a]` is followed while the element branch is not considered for traversal. Demanding WHNF for an expression `(spine_list some_list)` suffices to ensure that a complete traversal is done. For the more complex binary search tree of Example 3, a more complex spine function is needed:

```
spine_bst :: BST a -> ()
spine_bst E            = ()
spine_bst (Node l _ r) = spine_bst l `seq` spine_bst r
```

The `seq` operator has now to be used to ensure a complete traversal when only WHNF demand is exerted. This is because there is more than one recursive branch in the `Node` case. Using the generic approach presented earlier, the spine traversal can be generalised as shown in Figure 5.6. Three functions `spine0`, `spine1`, and `spine2` are given for the flat kinds `(*)`, `(* -> *)`, and `(* -> * -> *)`, respectively. Each function sequentialises the traversal of all branches of the given value using the auxiliary function `seqs`, which we introduce as an extension of the `seq` operator. Spine evaluation is very helpful for:

- spanning up lazily defined data structures which are needed anyway
- walking through a data structure of function (or process) applications to ensure timely execution (or process creation)

Within Eden one often deals with data structures which are first divided to form a collection of partitions before they are worked on by parallel processes. To form the connection between processes and partitions, one very often maps a process abstraction on the partition collection. A `spine` function is then normally used to start all processes at (almost) the same time via a traversal. As `spine` does not exert demand on interior elements, usually additional demand is contained within each node. The next subsection provides functions for doing that comfortably.

```
seqs :: [a] -> b -> b
seqs acts fin = foldr seq fin acts

spine0 :: Abs0 t => t -> ()                                  -- (*)
spine0 t = recs
  where (_,(ts,_)) = suco0 t
        recs = seqs (map spine0 ts) ()

spine1 :: Abs1 t => t a -> ()                                -- (* -> *)
spine1 t = recs
  where (_,((_,tas),_)) = suco1 t
        recs = seqs (map spine1 tas) ()

spine2 :: Abs2 t => t a b -> ()                              -- (* -> * -> *)
spine2 t = recs
  where (_, ((_, _, taas, tabs, tbas, tbbs),_)) = suco2 t
        recs = r taas `seq` r tabs `seq` r tbas `seq` r tbbs
        r x  = seqs (map spine2 x) ()
```

Figure 5.6: Generic spine traversal

In contrast to spine evaluation one also sometimes needs a full NF evaluation. For doing that, an evaluation function has to be defined and overloaded for every data structure which might occur during NF evaluation. This is necessary as in our approach generic functions do not descend down into a data structure but merely peel off the outermost type constructor layer. As shown before, overloading can be implemented via type classes or simulated via explicit type annotations in combination with GADTs (Figure 5.4).

Regarding nested data structures always only the spine of the *outermost type* is traversed by `spine`. Deeper-reaching spine traversals for nested structures like `[BST a]`, which traverse both list and tree structure and leave the `a` elements untouched, must be defined separately. This is reasonable, because one cannot tell how deep the spine has to reach and which part has to be left out.

5.5.2 Selective Evaluation

Spine evaluation does not touch the elements of a data structure and traverses the whole structure. In contrast to that we will now give the definition of a `touch` function (see Figure 5.7), which allows for discontinuing the traversal for branches determined by a given predicate. Furthermore a function for controlling demand exerted on element values can also be given. All this makes `touch` a general function for data-oriented demand control.

For a given value, `touch1` simply applies the element demand function `da` to all element values found at the root. Furthermore it uses the `c` function to filter out those branches for which traversal shall continue. `always` is a simple auxiliary

```
touch1  :: Abs1 t => (a -> ()) -> (t a -> Bool) ->
                     t a -> t a
touch1 da c t = as_eff `seq` tas_eff `seq` t
  where (_,((as,tas),_)) = suco1 t
        as_eff           = seqs (map da as) ()
        tas'             = filter c tas
        tas_eff          = seqs (map (touch1 da c) tas') ()

touch1n :: (Abs1 t1, Abs1 t2) =>
           (a -> b)            ->
           (t2 a -> Bool)      ->
           (t1 (t2 a) -> Bool) ->
           t1 (t2 a)           -> t1 (t2 a)
touch1n da c2 c1 = touch1 (touch1 da c2) c1

always :: a -> Bool
always _ = True

touch2  :: Abs2 t => (a -> ()) -> (b -> ()) ->
                     (t a a -> Bool) -> (t a a -> Bool) ->
                     (t a a -> Bool) -> (t a a -> Bool) ->
                     t a b -> t a b
touch2 da db c_aa c_ab c_ba c_bb t =
  as_eff `seq` bs_eff `seq`
  taas_eff `seq` tabs_eff `seq` tbas_eff `seq` tbbs_eff `seq` t
  where
    (_, ((as, bs, taas, tabs, tbas, tbbs),_)) = suco2 t
    taas'    = filter c_aa taas
    -- tabs',tbas',tbbs' analogously
    as_eff   = seqs (map da as) ()
    bs_eff   = seqs (map db bs) ()
    tabs_eff = seqs (map (touch2 da db c_aa c_ab c_ba c_bb) tabs') ()
    -- taas_eff,tbas_eff,tbbs_eff analogously
```

Figure 5.7: Generic touching with element demand and traversal predicate

function which can be inserted for `c` when a full traversal is desired. All effects are then sequenced as the result of the function. `touch2` is constructed analogously. Given these functions, `spine1` can easily be defined in terms of `touch1`:

```
spine1 = touch1 (\_ -> ()) always
```

`spine2` can be redefined analogously.

As usual in our approach to generic programming, `touch1` and `touch2` will only work on the outermost type constructor layer of the value to be demanded. If, however, deeper demand exertion has to be applied to nested data structures an appropriate nesting of `touch` functions has to be used. A version of `touch1` for two nested data structures is `touch1n`, which is also shown in Figure 5.7.

---

7 Example (Touching a binary search tree)
As an example for `touch` consider a function `span_bst` which forces evaluation of a given binary search tree. Elements are evaluated to WHNF and traversal of branches is determined by an element valuation function `f`. Note that the value inspection done by `f` will itself already cause evaluation of that element; therefore the `element_eval` function is mainly defined to ensure element evaluation of the first node.

```
span_bst :: (a -> Bool) -> BST a -> ()
span_bst f = touch1 element_eval cont
  where element_eval x    = x `seq` ()
        cont E            = False
        cont (Node l x r) = f x
```

5.5.3 Universal Functions for Four Reduction Degrees

To sum up, Table 5.8 shows the functions by which we can obtain the four reduction degrees shown in Figure 5.1 universally for all data structures. This range of functions allows for flexible demand steering on data structures.

Red. degree	Description	Function(s)
E0	Unaltered	`untouched _ = ()`
E1	WHNF	`whnf x = x `seq` ()`
E2	Spine traversal	`spine0`, `spine1`, `spine2`,
	Selective traversal	`touch1`, `touch2`
(E3)	NF	`NFData` class instances created in Chapter 3

Figure 5.8: Universal reduction functions

5.5.4 Annotated Data Structures

The functions shown so far allow for evaluation of data structures in various ways. When writing a program, data structures are carefully chosen with regard to access

complexities and memory requirements. However, some other properties of a data structure regarding its status during a program run are not always clear:

- Is it needed frequently and completely or only sporadically and partially? Is it intermediate and of a short life-time (garbage-collected soon)?
- Is it global within the program (passed through many functions) or only of local interest (not passed to an external function)?
- At a certain point in the program, which is its current status? To which degree has it already been evaluated? There is no point in traversing an already evaluated data structure a second time.
- In a parallel setting, is this data structure a partition of a bigger data structure? Is it used on one node only or is it sent to other nodes?

This kind of *meta information* is often available during the design process, used during the process, but finally not kept in the program. At a later point in time, this makes programs harder to understand and to modify, as these vital informations are not present anymore; program modifications are harder as they could be because demand control is very sensible and the proper amount of demand can easily be too big or too small, especially in a parallel setting. We therefore propose the *explicit annotation* of data structures on the expression level.

We will wrap selected data structures in the `Info` data structure shown in Figure 5.9. This data structure is a simple enumeration of properties and can easily be extended by additional ones. Instances of common classes are defined to hide the `Info` from all other operations. Within a program, the additional `Info` layer around a data structure is then ignored by functions aware of it or can explicitly be removed by applying the `stripInfo` function. We are aware of the newly introduced administration overhead; but if this annotation is used for all major data structures one can benefit from the given information.

Upon getting an argument data structure, a function can immediately inspect its general status like its neededness and whether it is a global or local structure (seen from a function's point of view). In a parallel setting one can also get information of this data structure will be communicated between nodes or whether it will stay on the local node. Additionally we will note if the data structure is part of a larger one. Another entry could be the type of the data structure; then `Info` could also cover the `HasType` data constructor of Figure 5.4.

The current evaluation degree of a data structure is described via `State` which records the current evaluation degree via one of four simplified degrees. This is harder to follow than all the other properties: As lazy evaluation employs sharing, duplicate references to the same data structure can easily be generated:

```
let datastructure = ...
in f (g datastructure) (h datastructure)
```

Both can now experience different evaluations, which nevertheless affect the same data structure. At such a branch the programmer has to make sure that the more extensive of both evaluations is stored in the `Info` table.

```
data Info a = Info {
                val    :: a,     -- The wrapped value
                fully  :: Bool,  -- True: Fully needed, False: Lazy
                global :: Bool   -- True: Global,       False: Local
                remote :: Bool   -- True: Remote,       False: Local
                part   :: Bool   -- True: Partition,    False: Top-level
                state  :: State  -- Evaluation state
               } deriving Show

data State = Untouched | WHNF | Spine | NF

stripInfo :: Info a -> a
stripInfo = val

instance Eq a => Eq (Info a) where
  (Info x _) == (Info y _) = x == y
  (Info x _) == (Info y _) = x == y
  (Info x _) == (Info y _) = x == y
  (Info x _) == (Info y _) = x == y

instance Ord a => Ord (Demand a) where
  compare (Info x _) (Info y _) = compare x y
  compare (Info x _) (Info y _) = compare x y
  compare (Info x _) (Info y _) = compare x y
  compare (Info x _) (Info y _) = compare x y
```

Figure 5.9: Data structure annotations

5.6 Control-Oriented Demand Steering

We have seen that data-oriented demand steering provides many possibilities to evaluate a data structure to different degrees. But how, when, and in which order does demand reach these isolated operations? Control flow, which is equal to demand flow in lazy languages, has to reach and touch these function calls to trigger evaluation. Accurately specified *local demand on data structures* is of no use if its activation is not equally accurate in a *global context*. Therefore we will look at control flow steering in this Section. But first we will introduce a common notion for *effects* executed on entering a function; we will start with an example of an effect.

5.6.1 Early Process Creation

Looking at the parallel computation of Hamming numbers back at Section 5.3.2 we have seen that for gaining real parallelism demand has to be considered. By applying Haskell's lazy semantics also to the additional Eden parallel constructs all what we gained was not real parallelism but only distributed sequentiality.

As discussed before[118] , the restrictive semantics for Eden constructs is therefore defined to allow the following:

- *A process can be created before its result is really needed.* This is an essential requirement for gaining real parallelism, as only processes whose runtimes overlap can be called parallel. In the following we will describe how this can be done.

- *Results are evaluated to NF.* This will keep processes producing results independently of explicit outer demand for these results. We are acting on the assumption that all results produced by processes are typically needed for the most part. One exception, which demonstrates the possible misbehaviour of this strategy, is a stream of trivial elements which can easily be computed. This may lead to a heap flooded by possibly unneeded elements. The next item shows an Eden mechanism for limiting that effect.

- *Results are transmitted eagerly.* This will keep the sending node sending result values as soon as they are available, saving explicit request messages. To avoid flooding the receiving node with unneeded result elements, the following mechanism has been incorporated: If a garbage collection on the receiving node collects the result data structure, a message will be sent to the sending node to terminate the result producing thread. This will also delete the reference to the remaining result values at the sending node which will cause their garbage collection. Partial results which are already on their way will be deleted on arrival.

To circumvent distributed sequentiality, processes defined via the `(#)` operator are behind the scenes created via the mechanism shown in Figure 5.10. A function containing a process instantiation `(p # x)` is transformed into a version which wraps (or lifts) the result value into a newly created data constructor `Lift`. The `(#)` operator is replaced by a slightly different function `createProcess`, which does not yield the result directly but wrapped into `Lift`. If `createProcess` is evaluated to WHNF, the process is created. The problem with distributed sequentiality was, that the control flow stopped waiting to obtain the WHNF of the result. Even getting the comparatively small WHNF of the result could take a long time (depending on the necessary evaluations); the solution therefore is to provide an artificial data constructor wrapping around the result which is immediately available and allows the control flow to continue. The regular occurrences of the result have of course each to be equipped with an additional `deLift` call to regain the original value.

The mechanism shown can also be automated to a certain amount. The *eager transformation*[143] is a `let` transformation which applies the mechanism to let-bound process applications like:

```
f x = let r = p # x
      in ... r ...
```

These are by definition considered as top-level and created speculatively. During program design one has to be aware of that transformation, as careless placements of process applications in let expressions within a recursive function could easily

```
f x = ... (p # x) ...
```

⇓

```
data Lift a = Lift a

deLift :: Lift a -> a
deLift (Lift x) = x

createProcess :: (Trans a, Trans b) => (Process a b) -> a -> Lift b

f x = let lifted   = createProcess p x
          delifted = deLift lifted
      in lifted `seq` (... delifted ...)
```

Figure 5.10: Early process creation by result lifting

lead to a flood of speculatively created processes (runaway parallelism). As shown before, this transformation can be done via the preprocessing scheme of Chapter 3.

If this mechanism is used consequently, distributed sequentiality gives way to real parallelism; this parallelism, however, is not yet optimal as control flow has still to reach every process creation site first. In the following, we will call process creation an effect and show that effect groups are desirable for parallelism.

5.6.2 Effects and their Execution

In a non-imperative language like Eden, where computation proceeds by evaluating functions and rewriting expressions, can there, besides the side-effects introduced by Eden's parallelism constructs, be something like an effect? Effects can of course be covered by monads, as it is usually done with the `ST` and `IO` monads or any self-written instance of the `Monad` class. But also on a *non-monadic level* we can identify another kind of effects in Eden: All these evaluations belong more to *how* something is computed than to *what* is computed, for example:

- *Process creation.* As seen before, to achieve early process creation the expression containing `createProcess` is given a name (`lifted` in Figure 5.10) and touched by the control flow upon entering the function. Due to sharing the name is then later a reference to the process result, in a sense similar to using a dynamic channel reference.

- *Local demand on data structures.* Local demand-steering techniques like the ones described in Section 5.5 are also effects as they are only steering mechanisms. They just prepare data structures for subsequent evaluations.

- *Unsafe I/O.* Small unsafe I/O operations not integrated into the I/O sequence starting from the `main` expression are also effects. For example, they include debugging message output like:

```
print :: String -> ()
print s = unsafePerformIO (putStr s)

f ... = print "Entering f" `seq` ...
```

- *Filling dynamic channels.* Another effect is the usage of a dynamic channel. If a channel name has been received it can be filled with data by calling `parfill`. This is usually done in the middle of an expression, which makes sense if it is placed within a case selection. If not, it can be pulled out of the expression:

  ```
  parfill cname value () `seq` ...
  ```

Seen from an aspect-oriented point of view it is desirable to separate these intra-functional effects from the remaining expression: The clear separation provides not only a quick overview of what effects are executed within a function, but the effects also usually benefit from being executed early.

Effects usually result in the unit value `()`, but may also yield a lifted value after a process creation or a data structure after exertion of demand. We therefore give three different, yet still simple, operators for sequencing effects:

```
(>.)  :: a -> b -> b                 -- Sequence
x >. y  = x `seq` y

(>:)  :: a -> (a -> b) -> b          -- Seq. w/ argument
x >: y  = x `seq` (y x)

(>:.) :: (Lift a) -> (a -> b) -> b   -- Seq. w/ arg. and delift
x >:. y = x `seq` (y (deLift x))

(>::) :: Abs1 t =>                   -- Seq. w/ arg. and generic delift
         t (Lift a) -> (t a -> b) -> b
x >:: y = x `seq` (y (gmap1 deLift x))
```

The first one just renames `seq` for clarity and sequences effects whose results are either irrelevant or the unit value. The second one (similar to the monadic `(>>=)` operator) allows for passing the result of the first effect as an argument to the next effect. This can be the case when demand is exerted on data structures and the evaluated data structure is used subsequently in another effect. The third one aims at postprocessing a process creation by additionally applying a `deLift` call to argument passed by the first effect. Lastly the fourth one assumes that the first effect generates a data structure of lifted values which is passed as an argument. This data structure is traversed, its are elements delifted, and the structure is used in the remaining effects.

Using these combinators we can now start to systematically extract effects out of a function and let them precede the function's result expression. Figure 5.11 uses a technique shown in Section 5.4 to wrap I/O and other effects around a function.

This means, that one can specify a group of effects which is always executed when *entering* the function and another group which is executed when *leaving* the

```
 f args = result

                                   ⇓

import System.IO.Unsafe

f args | beforeIO >. beforeEff >.
         whnf     >.
         afterIO  >. afterEff  >.
         False     = undefined
       | otherwise = whnf
       where beforeIO  = unsafePerformIO (putStr "Entering f ...")
             beforeEff = ()
             whnf      = f' args
             afterIO   = unsafePerformIO (putStr "... leaving f")
             afterEff  = ()
f' args = result<f/f'>
```

Figure 5.11: Wrapping I/O and effects around a function

function; to initiate that it suffices to reach the function call with WHNF. In between the original function `f` is called. `f` has been transformed to `f'`, in which all calls to `f` have been replaced by calls to `f'` via substitution:

5.5 Definition (Substitution)
We denote the textual substitution *of every* `x` *by* `y` *within* `e` *by* `e < x / y >`. *Previous proper renaming to avoid name clashes is assumed. To clarify the notation,* `(a+b)<b/c>` *would yield* `(a+c)`.

In the first of the two guards `f'` is called to demand WHNF evaluation; this guard is then (after executing all effects) rejected when `False` is encountered. The second guard then delivers the function result, which (by sharing) is already available in WHNF. Using this scheme for a function one can easily separate effects from the remaining computation.

The scheme also lends itself to automatic transformation, for example to annotate functions with debugging or tracing effects. Then one would choose to submit information like the ones shown in Figure 5.11 about entering and leaving the function (augmented by the specific argument values) to the console or some trace file. The latter one is harder to achieve because these I/O operations are isolated and it is not immediately clear where the file handle can be kept. However, one could give the same dynamic channel to every function and use the channel's merging behaviour for collecting unordered tracing results from all functions. This can be done if the restriction, that every dynamic channel may at most be used once, is lifted. Such transformations for the systematic introduction of effects can easily be implemented via the preprocessing mechanism shown in Chapter 3.

Besides wrapping the whole function in effects, one would often want a more accurate distinction when dealing with functions with alternatives. Alternative-specific effects can then simply be specified by wrapping each alternative in effects, like we have already shown for early process creation:

```
f alt1 = effs1 >. exp1
f alt2 = effs2 >. exp2
```

The whole function can of course also again be wrapped in effects like shown above.

---

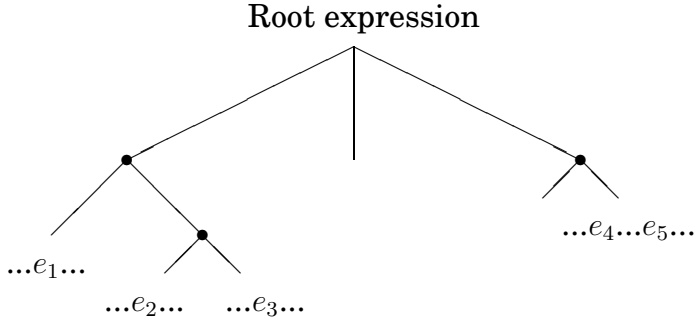
8 Example (Generic parallel map with effects)
All together we can formulate a generic parallel version of map in Eden which clearly separates effects from computation:

```
ggParMap1 :: (Abs1 t, Trans a, Trans b) =>
             (a -> b) -> t a -> t b
ggParMap1 f v = effects >:: id
  where effects = touch1 whnf always results
        results = gmap1 (createProcess (Proc f)) v
```

The function needs only an initial WHNF impulse to create the whole process structure.

5.6.3 Effect Groups

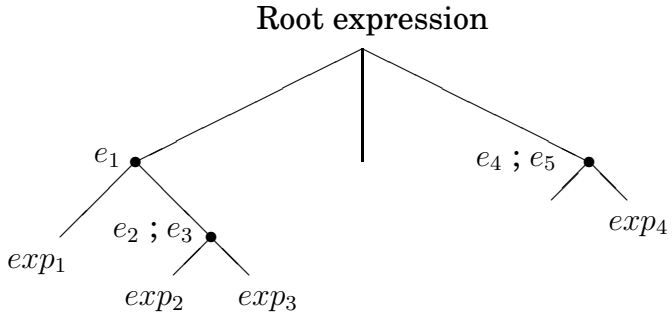
An expression can be seen as a tree, with effects, like the ones mentioned above, being scattered all over the tree. As an example for discussing effect distribution within a program we have sketched such a tree below for a fictional root expression with randomly distributed effects. Functions, marked here by black circles, are then subtrees. Some leafs are indeed subtrees themselves, these are displayed in a shorter form as an expression $...e_i...$ containing an effect e_i:



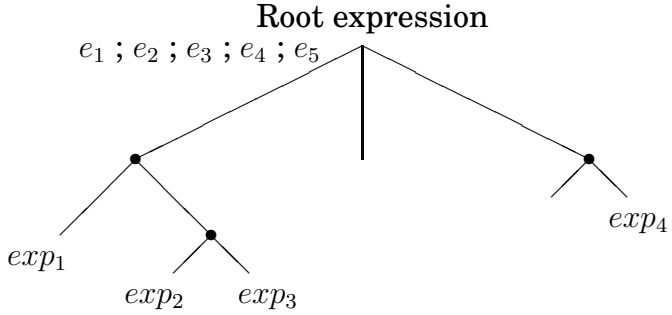
During execution, lazy evaluation will traverse the tree and trigger an effect when encountering it. This leads to an unpredictable and uncontrolled effect execution order. This is especially undesirable in the context of process creation in Eden, as this will usually yield only little parallelism due to only small (if any) temporal overlapping of processes. The execution order of the other effects is not

as critical, although a `parfill` can also benefit from early execution. Demand and I/O can, if they are to be executed anyway, just as well be executed early.

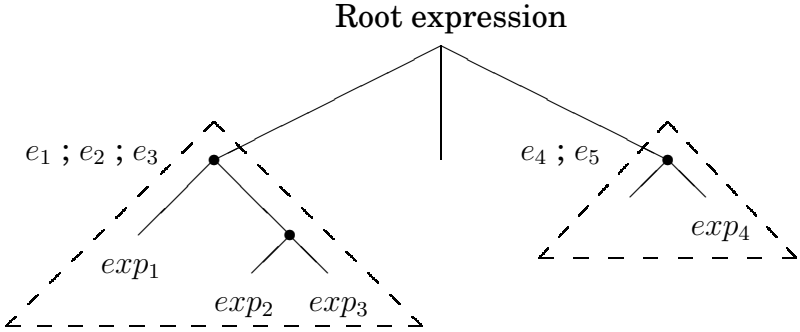
This is why *effect groups* are desirable. Especially process creations which are executed anyway can be pulled to the front of a function's main expression and be executed right after entry. This *local overlap* causes at least the processes started by the same function to have a common starting time. This can easily be achieved via the mechanism shown in Figure 5.11.



Overlap between different local groups ($\{e_1\}$, $\{e_2, e_3\}$, and $\{e_4, e_5\}$ in our example) is harder to increase. The most desirable solution would be to gather all local effects in a single, global group to achieve maximal simultaneity:



But this is of course hardly viable, as the scope and the visibility of identifiers used within the effects is limited to local declarations. One could pass these through all functions between the maximal effect group and the effect's home function, but this is very tedious and spoils the program code. Therefore one steps back to achieving a small set of local effect groups which are as big as possible.



And this is exactly what some skeletons are providing: For a subtree of the expression tree they collect all effects (in a data structure or on the control-level) and instantiate them at almost the same time in a structured way. If more than one skeleton is used within a program, one has then also to take care that these are triggered for maximum inter-skeleton overlap. This can be done via continuous demand control throughout the whole program.

5.6.4 Automatic Grouping

As it is a tedious task to extract effects out of a function and to form an effect group executed on function entry, we would like to have an automated mechanism for effect grouping. Using our meta-programming facilities of Chapter 3, we are able to define a kind of *automatic local effect grouping*. This transformation will collect the easy to spot process creations which are important for an Eden program's performance and move them to the function's entry point. Other effects are neglected, as they are either very much harder to identify or not already contained in the code and must be explicitly defined by the user. This grouping will only consider effects contained in a single function and will not group effects of different functions; to accomplish this, a much more extensive function analysis and transformation would be necessary. An automated grouping within a single function will face expressions or function declarations:

Regarding *expressions* there are two cases: *Firstly*, the process creation can be contained within a `let` block. Then the needed transformation is already contained in the Eden system and called the *eager transformation* which we have already shown in Chapter 3. As a reminder, written with our new operators and reduced to a single effect:

```
let r = p # a              let r' = createProcess p a
in e               =>      in r' >:. (\r -> e)
```

If other local declaration blocks are nested into the `let` declarations, these will not be considered top-level and therefore not be traversed. The effects contained within these will not be moved. *Secondly*, the process creation could be contained within the expression as an anonymous subexpression. The resulting transformation is equally easy (also only shown for a single effect):

```
(... (p # a) ...)  =>      let r = createProcess p a
                           in r >:. (\r -> ... r ...)
```

The last case can easily be integrated into the eager transformation.

Regarding *function declarations* there are also two cases: *Firstly*, the function's main expression is treated like shown above. *Secondly*, process creations can be contained within the local `where` declaration block. This case is treated like the `let` block above. Like above, nestings are not traversed.

While the expression transformation is automatic, the function transformation is optional. For selective application we can therefore introduce a Template Haskell splice which can easily be applied by the programmer to those functions whose top-level effects shall be grouped. The splice, which we will not show in detail, can be applied as follows:

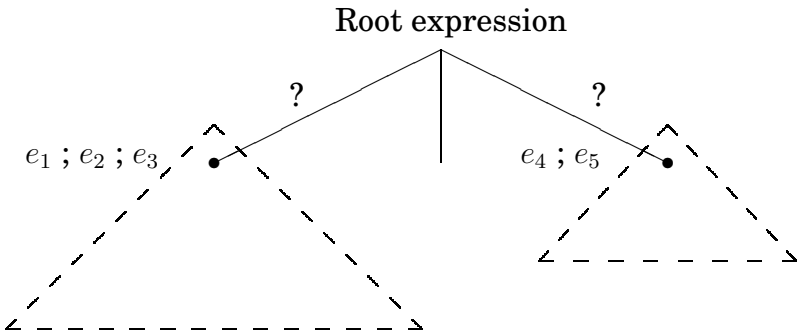
```
$(bundle [| f args = result |])   -- bundle annotation by programmer

bundle :: Dec -> Q Dec            -- predefined bundle function
```

It embraces a function out of which all top-level process creations are collected via traversing the function subtree. Calls triggering these effects will then be bundled at function entry.

5.6.5 Demand Management

Finally we have achieved (manually or semi-automatically) a program in which effects are gathered in local groups of considerable size. Touching a group with WHNF demand suffices to reliably trigger execution of the whole group. Now that we have the demand flow in every group under control, what about demand flow between groups? How can we ensure that every group is touched early, reliably, and almost simultaneously?



One traditional method is to carefully construct functions in such a way that everything just fits in the right place. Everything is laid out so that demand finds its way through the code to every group in time. If some difficult demand steering remains, then usually a punctual demand control, resembling a kind of hack, is inserted. The disadvantage is that all code parts are interdependent and that the whole setup is therefore very fragile. Modifications can easily destroy the intended evaluation order and depth. This happens even more easily as demand needs and exerted demand are almost never documented. Just having the undocumented code at hand it is hard to understand why at a certain point the correct amount of demand arises. Unexperienced programmers then often fall back to randomly inserting demand into a program to get it running in the desired way. Demand annotations would help, but demand informations are not kept in a program.

One would never think like that of type annotations. In the Hindley-Milner type system it is completely unnecessary to give explicit types, as each type can be inferred automatically. Often the given types are also inferior to those inferred, as the given types are very often much too special: When writing a function (and its type), one mostly has a single use of that function in mind and types the function according to that use. For example in a program which calculates with matrices and vectors one might write:

```
map :: (Matrix -> Vector) -> [Matrix] -> [Vector]
```

```
map f ms = [f m | m <- ms]
```

The special type is accepted, although the well-known more general type of `map` would have been much more adequate and not as misleading as the above type can be if this `map` is used in another context.

When types can be inferred automatically, and the inferred types are at least as exact as those given by the programmer, why does almost every programmer explicitly type each function? Because:

- During writing, it is helpful to clearly see a function's intended input/output behaviour. Type systems are good because they can point out mistakes during compilation; they are even better, as they encourage the programmer to write programs in a structured and incremental way by first defining the function's type before constructing the function.
- After writing, type annotations are not deleted although they could be. They are left in the program as they document the programmer's intention and aid in understanding the program if later modifications are to be done. The first thing usually read when first seeing an unknown function is its type.

Then why is this conviction not carried over to demand annotations? The same two points made above in favour of type annotations also apply to demand annotations. Therefore we propose explicit demand annotations as one of three measures to ensure a better *demand management* in a program:

1. Use effect groups (or skeletons) with strictly defined demand behaviour
2. Annotate each (or each central) function with its demand behaviour so that the progress of demand from the root expression, which is the source of all demand, to each effect group is clear
3. Ensure fully controlled propagation of demand from the root to each effect group

These three measures can be introduced into programming as sketched in the following:

1) Effect groups have been shown above. It is important to know which demand is necessary to start execution of the group and which demand is exerted to other expressions. This can be described via the notation sketched in the next item.

2) Demand annotations will not be introduced as an additional language construct but as special comments. Given a function we can describe its demand behaviour by noting which demand is exerted by the function on its arguments and internal function calls if a certain external demand exists.

Figure 5.12 shows a traditional definition of `parMap` with two auxiliary functions `tlList` (for forcing a list of process instantiations with subsequent delifting) and `forceWHNFSpine`. (Interestingly, the latter function even encodes a part of its demand behaviour in its name: It exerts WHNF demand on every list element and evaluates the list spine.) For each function we have

sketched a possible demand notation; within each demand description, the function's demand behaviour is shown for each of three possible external demands (WHNF, spine, and NF). Then for each argument the demand exerted on that argument is shown. Although being a bit long, the notation clears up a function's behaviour.

```
{ WHNF =>                Spine =>                NF =>
    a         : Nothing     a         : Nothing     a         : NF
    [.]       : Spine       [.]       : Spine       [.]       : Spine
    eagerIL   : WHNF        eagerIL   : Spine       eagerIL   : NF }
parMap :: (Trans a, Trans b) =>
          Process a b -> [a] -> [b]
parMap p xs = eagerInstList (repeat p) xs

{ WHNF =>               Spine =>              NF =>
    a       : Nothing     a       : Nothing     a       : NF
    Lift .  : WHNF        Lift    : WHNF        Lift .  : NF
    [.]     : Spine       [.]     : Spine       [.]     : Spine
    fWS     : WHNF        fWS     : Spine       fWS     : NF }
tlList :: [Lift a] -> [a]
tlList insts = forceWHNFSpine insts `seq` (map deLift insts)

{ WHNF =>         Spine =>        NF =>
    a    : WHNF     a    : WHNF     a    : NF
    [.]  : Spine    [.]  : Spine    [.]  : Spine }
forceWHNFSpine :: [a] -> ()
forceWHNFSpine []     = ()
forceWHNFSpine (x:xs) = x `seq` forceWHNFSpine xs
```

Figure 5.12: Example demand annotations

3) Demand propagation is then the next step: Starting with the I/O driven demand of the `main` expression, demand has to be passed on in a controlled way. Demand annotations show how functions propagate exerted demand to other functions. Together, functions can form a chain of demand propagation. Additionally for parallelism, as shown in Section 5.3.2 and solved in Section 5.6.1, it is also important to be able to *split demand*. This can in general be done by lifting as shown in Figure 5.10; then an immediately available outer result hull satisfies WHNF demand which can progress while a second demand chain has been started.

These three measures together enable the programmer to actively control and document demand flow in a program. Controlling demand has proven to be of great importance, both in a sequential (memory efficiency) and in a parallel (true parallelism) setting. The importance for parallelism is even higher, as Eden is a language that explicitly lays process control into the hands of the programmer and therefore relies on an equivalently powerful demand control. We have shown

mechanisms, some related to generic functions, and sketched a method for active demand flow control.

6. Data Parallelism

"Should I structure my program after the decomposition of the value it consumes ..."

Introduction of Meijer et al.[151] (continued in Chapter 7)

6.1 Motivation

Due to Fox[66] and Chapter 38.2.1 of Zomaya[217] , most parallel algorithms can be classified in terms of the regularity or irregularity of

- the underlying data structures and
- the synchronisation required as elements of these are updated.

These two data-oriented characteristics span four categories of parallel algorithms:

Synchronous. This is classical *data parallelism*. Regularly sized data elements are updated in a regular intervals. Algorithms of this class are usually associated to parallel machines of the SIMD class, and are ideally suited for being expressed in specialised data-parallel languages like High Performance Fortran (see Chapter 7 of Foster[64]). Algorithms of this category are quite predictable and can therefore be implemented quite efficiently.

Loosely synchronous. These algorithms deal with irregular data elements which are updated regularly. This means, that like before a global iteration or stepping scheme rules the computation, but that within each step the size of the data elements (and therefore the amount of work) can vary. To get an equal utilisation, usually some kind of load-balancing is therefore applied.

Embarrassingly parallel. These are very common and combine regular data structures with irregular synchronisation. Tasks are usually completely independent and are distributed via a static or dynamic load-balancing scheme.

Asynchronous. These algorithms lack regular synchronisation points as well as regular data structures and are therefore hard to express. They are classical examples of *control parallelism*.

In this chapter we will focus on data-parallel algorithms which include the first two categories. More easily expressed, data parallelism can also be understood as the parallelism one gets by applying the same operation to a group of data elements at the same time. A sequence of these steps is then a data-parallel program.

In Eden, data parallelism is quite important. Many functional data structures reveal parallelism in a natural way, and Eden makes it easy to exploit this parallelism. In this setting it is important to be able to partition the input data structure comfortably for distributing the parts of the data structure across the parallel nodes. We will show how to do that both in the traditional non-generic way but also using our generic programming approach. Having produced partitions, one usually works on these with the standard higher-order functions also known from the Bird-Meertens formalism [193] , like *map*, *fold*, or *filter*. In Eden parallelisations are often done via *map*, which can be expressed in different ways depending on its use. All non-generic parallel versions of *map* have generic counterparts, which we will also describe. Another problem with data parallelism in parallel functional languages is how one should deal with large data structures. We will show what can be done to avoid inefficiencies.

> Section 6.2 will deal with non-generic and generic methods for partitioning and regrouping data structures. Then Section 6.3 introduces non-generic parallel map skeletons into Eden which are generalised in Section 6.4.
>
> Parts of the chapter are based on three of our papers[119, 174, 141] .

6.2 Partitioning, Granularity, and Grouping

Partitioning, which is the division of a data structure in a structured collection of substructures, is an elementary part of parallel programming. In a *data-oriented parallelisation*, it is immediately clear that data has to be partitioned for distribution over the parallel nodes. The same is the case for *control-oriented parallelisations*, where each parallel task has to get its share of the input data structure on which the computation depends. It can also happen, that no noteworthy data structure is needed for a computation and that therefore a *data structure partitioning* is not necessary. Then one needs a *control partitioning*, which belongs to parallel algorithm design and which this section is not about.

Control partitioning is directly related to *granularity*. Normally, granularity is interpreted as the amount of work which results from evaluating an expression. The size and kind of a data structure partition directly influence the granularity of the computational task that is based on that partition. In a parallel setting it is especially important to choose the task grainsize carefully: Large sets of fine-grained tasks can most often be generated very easily, but due to high communication costs it is often not feasible to evaluate these in parallel. On the other hand, too coarse-grained tasks may leave nodes unemployed or induce stagnation in task distribution as there are not enough of them.

According to Sadayappan et al.[183] , granularity can also be measured as the

ratio of the compute time needed for task execution to the communication cost incurred by executing that task. Therefore granularity grows with the amount of work contained in a task, while it is considered shrinking if that is accompanied by lots of communication. The most useful grainsize depends on the number and the speed of the parallel nodes and the speed of the interconnection network. It is important for a parallel program to be able to change granularity depending on the parallel system on which it is run. As granularity directly interacts with partitioning, we will now set out to describe methods for data structure partitioning.

Before doing so we have to note, that not every partitioning function needs to have an inverse function for recombining the partitions. At times it may be acceptable to neglect the initial context as results are collected in another form.

6.2.1 Non-Generic Partitioning

Non-generic partitioning focuses exclusively on dividing explicit data structures like lists. We will not deal with the partitioning of implicit data structures like coordinate ranges, which can easily be split and administrated. Figure 6.1 shows two typical list partitioning methods; being standard collections, lists are a typical aim for partitioning. Both deal with partitioning a list into a set (also a list) of sublists. For each partitioning function f an inverse function f' is given such that $f' \cdot f = id$ holds. Both are reflected by the type synonyms `Partitioning` and `Combination`.

The first partitioning scheme takes a partition size n and cuts of pieces of that size starting from the beginning of the list. That implies, that the last piece has size $\leq n$. This means, that it could be considerably smaller than n, which can be a disadvantage during partition distribution. The second partitioning scheme makes for a better distribution by using the Bresenham distribution [34] . It divides the input list into sublists such that the difference between the lengths of any two sublists is always ≤ 1.

Figure 6.2 shows a most simple list partitioning on the element level. The list is essentially a tree, as every data structure in Haskell. It is weighted to the right such that its elements are attached to a spine of cons nodes which ends with the nullary nil constructor. The list can therefore be partitioned quite easily by travelling down the spine, counting elements, and filling in nullary nil contructors for the disconnected cons nodes. The collection of partitions can then (if partition order is maintained by the collection) easily be recombined because every partition contains only a single nil node. The recombination is done by traversing the first partition until the nil node is found; then the nil is replaced by the next partition which is then treated in the same way until all partitions have been added. This is not quite as easy in a generic setting, which we will treat now.

6.2.2 Generic Partitioning

In this subsection we will restrict ourselves to datatypes of kind `(* -> *)`; other flat kinds can be handled analogously. Now we will at first distinguish between destructive and non-destructive partitionings:

```
-- Types -----------------------------------------------------
type Partitioning  a =  a  -> [a]
type Partitioning' a = [a] ->  a

-- Direct partitioning ---------------------------------------
direct :: Int -> Partitioning [a]
direct _ [] = []
direct n xs = ys : direct n zs
  where (ys,zs) = splitAt n xs

direct' :: Partitioning' [a]
direct' = concat

-- Variable partitioning -------------------------------------
variable :: Int -> Partitioning [a]
variable n xs = (reverse . snd) (foldr f (xs,[]) ls)
  where ls          = bresenham (length xs) n
        f l (xs,ps) = let (p,xs') = splitAt l xs
                      in (xs',p:ps)

        bresenham :: Int -> Int -> [Int]
        bresenham total parts = take parts (calc total)
         where calc t = let (d,m) = t `divMod` parts
                        in d : calc (total + m)

variable' :: Partitioning' [a]
variable' = concat
```

Figure 6.1: Direct and variable partitioning methods (chunking)

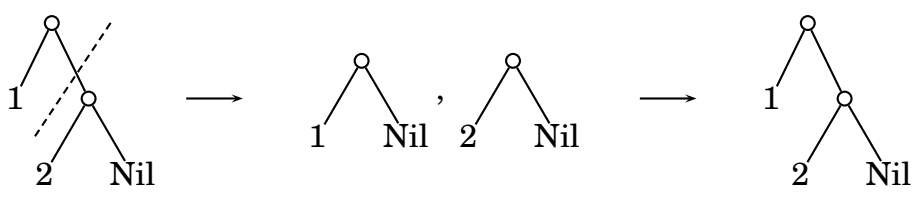


Figure 6.2: List partitioning on element level

```
prefix, postfix :: Abs1 t => t a -> [a]

prefix t = as ++ rest
  where (_, ((as, tas), rec)) = suco1 t
        rest                  = concat [prefix ta | ta <- tas]

postfix t = rest ++ as
  where (_, ((as, tas), rec)) = suco1 t
        rest                  = concat [postfix ta | ta <- tas]
```

Figure 6.3: Destructive generic partitioning

- *Destructive partitionings* extract elements out of data structures in a well-defined way and store these in a collection which will *not* allow a later reconstruction of the initial structure. This can be useful when the order of the extracted elements contains enough information and the initial structure is not needed for computing the result. These functions have the generic type:

  ```
  p  :: Abs1 t => t a -> [a]
  ```

- *Non-destructive partitionings* on the other hand deconstruct a data structure into parts of the same type, which can be recombined later. This is an often applied kind of parallelisation: Partition into smaller parts, treat these in parallel, and recombine the results. The generic types comprise a partition and a recombination function:

  ```
  p  :: Abs1 t =>  t a  -> [t a]    -- Abs1 t => Partitioning  (t a)
  p' :: Abs1 t => [t a] ->  t a     -- Abs1 t => Partitioning' (t a)
  ```

Non-destructive partitioning schemes can in principle also be used destructively. But additionally functions like the ones in Figure 6.3 can be used to extract elements in various fashions. The figure shows functions for a prefix and a postfix element collection; this collection can the be further partitioned with the functions of Figure 6.1. Level-wise traversal corresponds to the `flatten1` function shown at the end of Section 4.6.2.

Truly non-destructive partitionings are harder to define generically. We now have to be able to deal with arbitrary tree structures, for which general meaningful partitioning schemes have to be defined. Dividing them is also difficult, as it is not guaranteed, that a nullary constructor for filling open ends (which are created when the data structure is split into portions) is available. As an example, consider the partitioning of a rose tree

```
data Rose a = Rose a [Rose a]
```

which has no explicit nullary constructor. If several partitions are cut out of the whole structure, how can the open ends be filled? Even if the data structure, like a binary tree with empty leafs

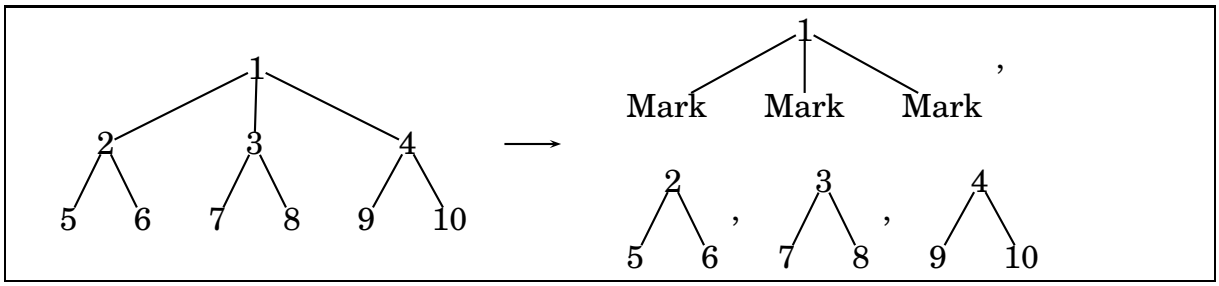


Figure 6.4: Tree partitioning on element level

```
data BinTree a = Leaf
               | Node a (BinTree a) (BinTree a)
```

possesses an explicit nullary constructor, it cannot be used for closing and marking the site of fracture, as it is not unique. Therefore we have to add a special nullary constructor to data structures to allow for non-destructive partitioning, if the order of partitions is otherwise not unique or no nullary constructor is present at all. The rose tree above lacks an explicit nullary constructor which we add now:

```
data Rose a = Rose a [Rose a]
            | Mark
```

Then the partitioning of Figure 6.4 is possible, where the mark defines a place where a substructure can be attached. For recombination, the first partition is traversed levelwise. If a mark is encountered, the next partition is attached at that place. The continued levelwise traversal will then yield the next mark; this is done until all partitions have been attached.

Having found a possibility to divide and recombine a data structure, this raises the question which general partitioning strategies make sense for arbitrary data structures? Data-parallel languages like High Performance Fortran offer combinable block, cyclic, and wildcard operators (see Foster[64]) for partitioning matrices flexibly. Can we do something similar for tree-shaped data structures?

We can define meaningful generic partitionings, which we will do in the following. At first we present in Figure 6.5 the function `gsingle1`, which implements the generic level-wise cut operation already sketched in Figure 6.4. The level can be chosen freely and yields the whole unchanged value for level 0. The cut in Figure 6.4 then represents a cut on level 1. The function works by descending down the data structure while rebuilding it from recursive results at the same time; if the correct level has been reached, the recursive subtrees are cut off, collected, and replaced by nullary mark constructors which have to be explicitly contained in the data structure.

---

9 Example (Single-level partitioning)
Partitioning a rose tree

```
example = Rose1 1 [Rose1 2 [Rose1 4 [], Rose1 5 []], Rose1 3 []]
```

on different levels yields the results:

```
gsingle1 :: Abs1 t =>
            Int -> Partitioning (t a)
gsingle1 0     t = [t]
gsingle1 level t = self : rest
  where (self, rest)  = gp1 0 t
        Just nc       = null1
        gp1 current x | current == level = (nc, [x])
                      | otherwise        = (rec (as,as'), concat tas')
          where (_, ((as, tas), rec)) = suco1 x
                (as',tas') = unzip [gp1 (current+1) ta | ta <- tas]

gsingle1' :: (Eq (t a), Abs1 t) =>
             Int -> Partitioning' (t a)
gsingle1' maxl ps = head (foldl gs1' ps [0..maxl])

gs1' :: (Eq (t a), Abs1 t) =>
        [t a] -> Int -> [t a]
gs1' (part:parts) level = trav 0 part parts where
  Just nc = null1
  trav _ val [] = [val]
  trav current val (p:ps)
    | current == level = if val == nc then (p:ps) else (val:p:ps)
    | otherwise = (rec (as, reverse tas'):ps_rest)
    where
      (_, ((as, tas), rec)) = suco1 val
      (tas', ps_rest) = foldl f ([], p:ps) tas
      f (acc,pas) ta = let (a:b) = trav (current+1) ta pas in (a:acc,b)
```

Figure 6.5: Non-destructive generic partitioning: single level-cut

```
gsingle1 0 example:
  [Rose1 1 [Rose1 2 [Rose1 4 [], Rose1 5 []], Rose1 3 []]]

gsingle1 1 example:
  [Rose1 1 [Mark,Mark],Rose1 2 [Rose1 4 [],Rose1 5 []],Rose1 3 []]

gsingle1 2 example:
  [Rose1 1 [Rose1 2 [Mark,Mark],Rose1 3 []],Rose1 4 [],Rose1 5 []]
```

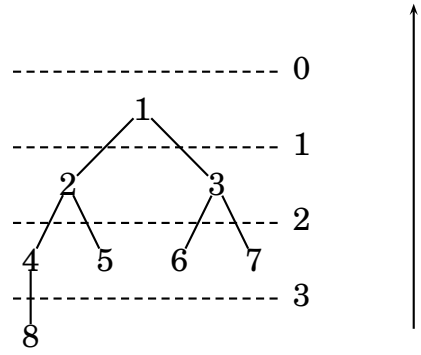
The dual function `gsingle1'` uses an auxiliary function `gs1'`, which takes a partition list and a level number as arguments. The first partition is used as the root out of which the complete data structure is reconstructed. It is traversed via a fold and all marks on the specified level are replaced by partitions in the partition list. The relative order of replacement is kept to level-wise despite a depth-first traversal; this is done via the level restriction.

```
gmulti1 :: Abs1 t =>
           Int -> (Int -> Int) -> Partitioning (t a)
gmulti1 end offset t = foldl f [t] levels
  where levels = takeWhile (>=0) (iterate offset end)
        f []     _ = []
        f (v:ps) l = parts++ps
          where parts = gsingle1 l v

gmulti1' :: (Eq (t a), Abs1 t) =>
            Int -> (Int -> Int) -> Partitioning' (t a)
gmulti1' end offset parts = head (foldl gs1' parts levels)
  where levels = (reverse . takeWhile (>=0)) (iterate offset end)
```

Figure 6.6: Non-destructive generic partitioning: multi level-cut

The single-level cut can easily be extended to a multi-level cut, as is shown in Figure 6.6. The function `gmulti1` takes two level arguments, the downmost level to start from and an offset function parameter. It then successively cuts the value into level-wise pieces of the offset calculated by applying the offset function to the current level. The function essentially folds the value upwards using the single-level cut as the folding function. Figure 6.7 shows this for an exemplary tree with a constant offset function `(\x -> x-1)`. `gmulti1'` is the corresponding function for




```
[parts0,parts1,parts2,parts3]

[v'''',parts1,parts2,parts3]

[v''',parts2,parts3]

[v'',parts3]

[v']
```

Figure 6.7: Multi-level generic partitioning

recombining partitions of a multi-level cut. Similar to `gmulti1`, it folds the reverse list of levels using the single-cut combination function `gs1'`.

Using these level-cuts one can already form many partitionings, including the usual list and tree partitionings. As said before, one has of course to restrict oneself to data structures which are defined inductively via data constructors. Haskell arrays or data structures hidden behind foreign language interfaces can of course not be handled that way. Also implicit data structures like a list represented by a range `(Int, Int)` has to be partitioned manually.

6.2.3 Grouping

We have used different partitioning methods to divide data structures into manageable pieces of little or even no interdependence. But do they fit the reality of the parallel computer on which the program will be run? Usually one defines too fine-grained pieces rather than too coarse-grained ones. This results in having too many pieces of work for the number of available parallel nodes. Therefore usually a static regrouping of pieces depending on the execution environment and a data distribution strategy is useful. This can be done using the meta-programming methods of Section 3.5.2.

The grouping itself can be done via the `combine` function

```
combine :: (a -> Bool) -> [a] -> [[a]]
combine p []          = []
combine p (x:[])      = [[x]]
combine p (x1:x2:xs) | p x1 == p x2 = (x1 : h) : t
                     | otherwise    = [x1] : combine p (x2:xs)
                     where (h:t) = combine p (x2:xs)
```

which creates groups in a list of elements depending on a predicate. During list traversal, elements are put in one group as long as the predicate holds. If it is no longer true, a new group is started until it holds again. This function can be used to regroup a list of partitions for mapping them efficiently on a parallel computer represented as a Haskell value in the style of Figure 3.18. This way, one can place data on the right level of the memory hierarchy and support locality.

6.3 Parallel Maps

The inherent parallelism of functional programs is usually quite fine-grained. It mostly reveals itself in the form of independent subexpressions which are strictly needed. The effort to evaluate these can then actually range from very small to very big. On the other hand, current parallel computers consist of fast computing nodes which are connected by comparatively slow network connections; this combination needs coarse-grained tasks to justify the large costs of task and result transportation compared to the task evaluation time. Therefore, the natural parallelism of a functional program has to be concentrated to form larger tasks. Grainsize is only one of many facets of parallel control which are vital for gaining speedups. Who should take care of these? There are two extremes between which Eden strikes a balance:

- *Implicit* approaches do not bother the programmer with parallel control and rely on program analysis for detecting sources of parallelism. This is difficult and may lead to poor parallelism.
- *Explicit* approaches burden the programmer with full control over processes, granularity, communication, and data distribution. Given such control, efficient programs can be constructed with some effort.

Data parallelism in Eden is often represented by the *map* function. This function can be called an *algorithmic skeleton*, as it captures a general pattern of computation on a high-level in a reusable form. Programming with algorithmic skeletons raises the abstraction level of the program, as these predefined building blocks usually have well-known properties and structure the program. For each algorithmic skeleton a fully controllable, parallel *architectural skeleton* can be predefined, which is tailored for different parallel computer architectures. These aim at exploiting special characteristics of a given architecture and are thus quite specialised and low-level. Therefore, programming with these specialises a program too much to a single architecture.

Like with parallel control, an intermediate level of abstraction has to be found which provides sufficient control to gain efficiency while sparing low-level details like process management. In Eden the situation is solved by introducing an intermediate layer of *implementation skeletons*[119] between algorithmic and architectural skeletons. These describe different Eden parallelisation methods of an algorithmic skeleton and are powerful and comfortable at the same time. They retain architecture-independence in the sense of GpH[206] , as the Eden implementation is based on a standard parallel message-passing library and quite easily adapts to new parallel environments. Taking *map* as a classical data-oriented algorithmic skeleton, we will now present four implementation skeletons. We classify the first three as data-oriented, while the last one is more control-oriented.

The standard parallelisation is called *parMap*, which is a direct parallelisation by evaluating each list element within a separate process; processes are usually distributed over the available nodes in a round-robin fashion, independent of the underlying parallel architecture. Driven by three possible savings three additional map parallelisations are induced:

- With a parMap, list size can easily exceed the number of parallel nodes. This results in multiple processes, which differ only in their argument, residing on the same node . The solution is here to group tasks so that the number of task collections is identical to the node number. Within such a *farm*, every node executes only one process.

- In a farm, the root process (or the calling main function) has to partition and group the arguments for the element processes. While no global load has yet been established, this is an inherently sequential action keeping the element processes from starting their evaluations (besides the ones for which the argument is not needed) and damaging speedup. Furthermore, as communication is expensive, the solution is to include the full argument in the process abstraction so that each process receives everything and chooses its part on its own. This replaces communication by recomputation and shifts work down in the process tree (which is always desirable). This pays off especially on systems with high communication costs. We call this the *direct mapping* (or communication fusion) approach.

- The third saving deals with process creation, which is expensive. The solution in a changing process structure can be to recycle processes. This means that a set of flexible processes is generated, which can change their functional

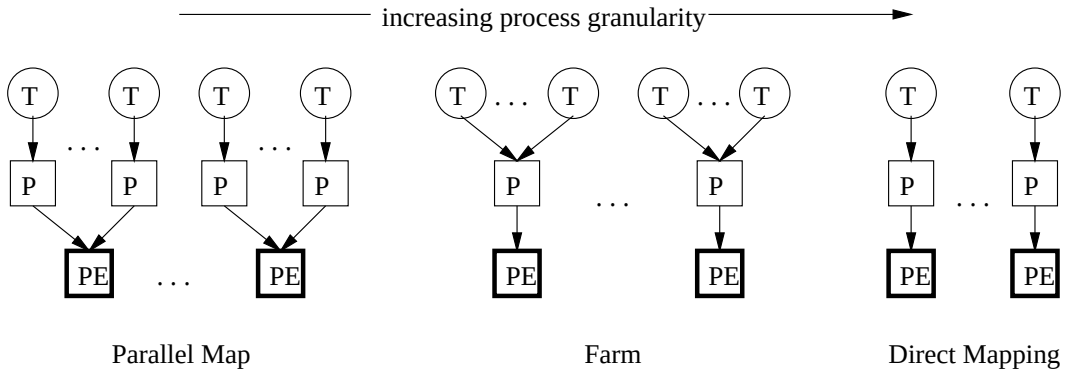


Figure 6.8: *map* implementation skeletons for regular granularity

behaviour and thus avoid process termination and recreation. A typical map version tailored for dealing with irregularly sized tasks is the *workpool*, which creates a set of worker processes which deal dynamically with different arguments. This version is considered control-oriented and is treated in detail in Section 7.3.

Figure 6.8 (taken out of our paper[119]) shows for each implementation skeleton how tasks are assigned to processes, and how processes are assigned to parallel nodes (also called processing elements or PEs for short). The typical non-generic implementations of these schemes for lists are shown in Figure 6.9. Every scheme has in the map type `Map a b` in the end for providing the same interface. The function `eagerInstList` is used in every scheme directly or indirectly for eager process creation. It takes care for early process creation as described earlier in Section 5.6.1.

`parMap` is then just an application of `eagerInstList` with a process list created by wrapping the function `f` in a process abstraction and repeating it to match the argument list. `farm` introduces partitioning and is a function composition which first partitions, computes via `parMap`, and recombines. `dm` uses processes with a void input parameter and input provided as a part of the process abstraction. Each process then extracts for itself its relevant part of the input data structure.

Having shown the approaches to parallelise *map* for lists, we will now set out to transfer these schemes to generic ones. These have to be able to use the generic partitioning and generic demand control schemes shown before.

6.4 Generic Parallel Maps

In the following we will apply our generic programming approach to the parallel functional language Eden.

6.4.1 Generic Skeletons

In the last subsection we have generalised representatives of four function categories. For which of these exists a meaningful version in a parallel setting? Clas-

```
type Map a b = (a -> b) -> [a] -> [b]

eagerInstList :: (Trans a, Trans b) => [Process a b] -> [a] -> [b]
eagerInstList ps xs = tlList insts
  where insts = zipWith (createProcess) ps xs

        tlList :: [Lift a] -> [a]
        tlList insts = forceWHNFSpine insts `seq` (map deLift insts)

        forceWHNFSpine :: [a] -> ()
        forceWHNFSpine [] = ()
        forceWHNFSpine (x:xs) = x `seq` forceWHNFSpine xs

-- 1) parMap ------------------------------------------------------

parMap :: (Trans a, Trans b) =>
          Map a b
parMap f inputs = eagerInstList (repeat (process f)) inputs

-- 2) farm --------------------------------------------------------

farm :: (Trans a, Trans b) =>
        Int                       ->   -- number of available nodes
        (Int -> Partitioning  a) ->   -- partition
        (Int -> Partitioning' b) ->   -- recombine
        Map a b
farm np part part' f inputs =
  ((part' np) . (parMap (process f)) . (part np)) inputs

-- 3) direct mapping ----------------------------------------------

dm :: (Trans a, Trans b) =>
      Int                       ->     -- number of available nodes
      (Int -> Partitioning  a) ->     -- partition
      (Int -> Partitioning' b) ->     -- recombine
      Map a b
dm np part part' f inputs = part' results where
  results            = eagerInstList proclist (repeat ())
  proclist           = [ proc (extract i np inputs) | i <- [0..(np-1)] ]
  proc arg           = process (\() -> f arg)
  extract i n inputs = (part n inputs)!!i
```

Figure 6.9: Three non-generic implementation skeletons for *map*

sical parallel skeletons[119] are often derivatives of *map*, therefore we show their generalised counterparts in Figure 6.10:

- `gParMap1` uses the generic `gZipWith1` to create a data structure of process instantiations which are demanded by a generic `spine1` traversal function. Demand causes early process creation which supports maximum parallelism.
- `gFarm1` and `gDirect1` are closely related to `gParMap1`. While `gParMap1` typically creates a process for every element of its argument data structure and places more than one process on each node of a parallel machine, both `gFarm1` and `gDirect1` let each process work on a whole `partition` of their argument. This is usually done so that only one process resides on each node. A `join` function later combines the results. The difference between both functions is that within `gFarm1` the function calling `gFarm1` is doing the partitioning for all processes created.
- `gDirect1` on the other hand gives each process the complete argument (spacesavingly housed within the process abstraction) for partitioning within the process.

As an aside, we show that the demand control function `spine1` of Figure 5.6 can also be expressed via `gfoldr1`. This results in a shorter and clearer implementation.

How can these generic skeletons be used in a profitable way? Some advantageous uses are:

- These generic parallel maps enable the programmer to easily apply a function to every element of an arbitrary data structure in parallel.
- Obviously a `gParMap` is very flexible in its application. Its type (see Figure 4.5) shows that it essentially is able to take any data structure containing a type `a` and to replace all occurrences of that type by a value of type `b` corresponding to the argument function. This is related to the boilerplate approach[123] . This makes it a useful tool in parallel programming, where one often changes process structures and data partitionings.
- As shown before[118] , it is crucial for an efficient parallelisation to overrule laziness to some extent by placing additional early demand on process instantiations. This is usually done via data structure dependent sequences of `seq :: a -> b -> b`, which delivers its second argument and evaluates its first argument to weak head normal form. An often used representative is the `spine` function for lists, shown in Figure 6.10 as `listspine` inside a comment. With the generic definition of `spine1` the tedious and errorprone rewriting of that functionality for different data structures is not needed anymore.

But are there disadvantages? Due to the de- and construction of data constructors generic functions introduce significant additional overhead which can be costly when used frequently. With generic parallel skeletons however, these costs are much lower as generic functions are used only for creating process structures which, compared to data structures within other computations, usually contain only a comparatively small number of processes. Additionally these skeletons are

```
-- (* -> *) -------------------------------------------------------------
gParMap1 :: (Abs1 t, Trans a, Trans b) =>
            t (Process a b) -> t a -> t b
gParMap1 ps vs = let insts = gZipWith1 createProcess ps vs
                 in spine1 insts >. gmap1 deLift is

gFarm1 :: (Abs1 t, Trans a, Trans b) =>
          m -> (m -> t1 -> t a) -> (m -> t b -> t2) ->
          Process a b -> t1 -> t2
gFarm1 mode partition join p t = let rs = gParMap1 procs parts
                                 in join mode rs
  where parts = partition mode t
        procs = gmap1 (\_ -> p) parts

gDirect1 :: (Abs1 t, Trans (t a), Trans (t b)) =>
            m -> (m -> t a -> t (t a)) -> (m -> t (t b) -> t b) ->
            (t a -> Process () (t b)) -> t a -> t b
gDirect1 mode partition join p t = let rs = gParMap1 procs units
                                   in join mode rs
  where parts = partition mode t
        procs = gZipWith1 ($) (gmap1 (\_ -> p) parts) parts
        units = gmap1 (\_ -> ()) parts

spine1 :: Abs1 t =>
          t a -> ()
spine1 = gfoldr1 f z
  where f _ tas = foldr seq () tas
        z _     = ()

-- (* -> * -> *) -------------------------------------------------------
gParMap2 :: (Abs2 t, Trans a, Trans b, Trans c, Trans d) =>
            t (Process a c) (Process b d) -> t a b -> t c d
gParMap2 ps vs = let insts = gZipWith2 createProcess createProcess ps vs
                 in spine2 insts >. gmap2 deLift deLift insts

spine2 :: Abs2 t =>
          t a b -> ()
spine2 t = recs where
  (_, ((_, _, taas, tabs, tbas, tbbs),_)) = suco2 t
  recs = foldr (>.) () (concatMap (map spine2) [taas,tabs,tbas,tbbs])
```

Figure 6.10: Selected parallel generic functions

mostly called only once: As process creation is quite costly, static process systems are preferred over dynamically changing ones. On the other hand it is clear, that some overhead is still present. Also, the extended type context and the different type classes for different kinds have to be noted.

6.4.2 Combining Generic Skeletons

If we express `gParMap1` of Figure 6.10 in another way so that its type signature resembles that of `map`, we can even pinpoint the parallelisation within a nested data structure via higher-order functions; this (and more) is possible, because our generic functions are first class citizens (in contrast to other approaches):

```
gPMap1 :: (Abs1 t,Trans a,Trans b) =>
          (a -> b) -> t a -> t b
gPMap1 f v =
  let is = gmap1 (createProcess (Process f)) v
  in spine1 is ‘seq‘ gmap1 deLift is
```

Now we can easily define generic skeletons for inner and outer map-like parallelisations:

```
inner1 :: (Abs t1,Abs t2,Trans a,Trans b) =>
          (a -> b) -> t1 (t2 a) -> t1 (t2 b)
inner1 f = gmap1 (gPMap1 f)

outer1 :: (Abs1 t1,Abs1 t2,
           Trans (t1 a),Trans (t1 b)) =>
           (a -> b) -> t1 (t2 a) -> t1 (t2 b)
outer1 f = gPMap1 (gmap1 f)
```

The `inner` function can be very helpful if one has to deal with large data structure partitioned into a structure containing the parts, for example a list of matrix blocks. Using `inner` one can easily work on the partitions in parallel.

6.4.3 Example

We will now use our approach to generic programming for the successive improvement of a partitioning scheme used in a parallel program to portion an argument data structure into computationally equal-sized chunks. This is very important for gaining load-balance. Partitionings are typically often subject to change during parallel program development.

As an example we will use a simple raytracer for calculating a two-dimensional photo-realistic image of a three-dimensional scene[119] . For calculating the result image, for each coordinate of the image a ray is shot into the scene. The central function

```
trace :: Scene -> Coord -> Color
```

calculates for the given scene and the coordinate the coordinate's color by tracing the way the ray takes through the scene. This happens backwards starting at the

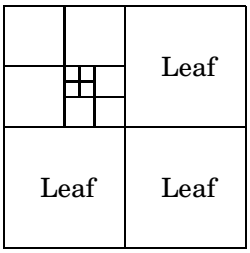


Figure 6.11: Exemplary quad tree partitioning

eye of the observer and ending at a light source. The image can then roughly be described by:

```
img = [trace s c | c <- allCoords]
```

When parallelising that algorithm, one can obviously compute each coordinate independently. As this results in far too fine granularity, one has to to coarsen the granularity by dividing the coordinate space into larger parts. A first improvement is to consider whole rows instead of single coordinates which allows us to parallelise via the given expression:

```
rows :: [ [Coord] ]
img = flattenRs (outer1 (trace s) rows)
```

To coarsen the granularity even more lists of rows can be grouped to form chunks:

```
rows :: [ [[Coord]] ]
img = flattenCs (outer1 (gmap1 (trace s)) rows)
```

When changing the chunk representation to

```
rows :: [ Set [Coord] ]
```

the old `img` function still works. Finally we will introduce a more complicated partitioning. As computational complexities vary greatly over the coordinate space, the partitionings shown above will usually not produce partitions of similar complexity. A better way is to divide the two-dimensional coordinate space into differently sized areas by using a quad tree partitioning:

```
data Quad a = Leaf a
  | Node (Quad a) (Quad a) (Quad a) (Quad a)

instance Abs1 Quad where
  null1                = Nothing
  suco1 (Leaf a)       =
    ("Leaf", (([a],[]), \([a'],_) -> Leaf a'))
  suco1 (Node a b c d) =
    ("Node", (([], [a,b,c,d]),
    \([],[a',b',c',d']) -> Node a' b' c' d'))
```

Figure 6.11 shows a simple quad tree partitioning where each box represents a leaf and contains a list of all coordinates of that area. As each box represents about the same estimated amount of computational complexity, the finer division of the upper left corner shows that more complex calculations are expected for that area. The

code used so far can even now easily be adapted for the new partitioning scheme:

```
quad :: Quad [Coord]
img = flattenQ (outer1 (trace s) quad)
```

In all cases the existence of functions for flattening the partitioning back to a list representation has been preassumed. The use of generic skeletons has enabled us to quickly switch partitioning schemes. Without generic skeletons a lot more work would have arisen.

7. Control Parallelism

"... or after the computation of the value it produces?"

Introduction of Meijer et al.[151] (continued from Chapter 6)

7.1 Motivation

In an Eden program, *parallel control* can be realised in different dimensions:

- *Small-scale* control comprises the use of single process applications created in different locations of a program without being organised in a joint context like a higher-order function. Due to the lack of encapsulation it is hard to create and coordinate parallelism, as processes are hard to relate to each other to gain simultaneity and thus parallelism. Therefore, such an approach is rare in Eden.

- *Large-scale* control is usually captured in a higher-order function and encapsulates a pattern of parallelism together with a defined strictness behaviour. As the activity of data parallelism is limited by data, the activity of control parallelism is limited by state. Therefore most often an explicit (or sometimes implicit) state is involved, which steers termination. As usually a single process keeps that state and decides on termination, a method to determine the current global state is needed. Often this can be done by observing and interpreting communication, but at times it may nevertheless be necessary to include explicit termination detection algorithms in the sense of distributed programming[146] . Large-scale control can differ in its nature:

  - A major difference is interaction: Most Eden programs are *transformational*, as they transform input values to output values representing a computation. *Reactive* systems[116] , however, are creating a system of interacting processes connected by lazy streams, ruled by state, and reacting to events. They are not easy to express within functional programming languages, as their specification requires complex mutual recursion and stream interactions.
  - Related to the former is the difference between distributed systems and parallel systems. While the former aims at creating stable, long-term

process systems on heterogeneous nodes accepting long latencies, the latter is aiming at executing a computation in the shortest possible time on a homogeneous system with low latencies. Both require extremely different methods of control parallelism.

- Task grainsizes, which can vary between regular and irregular, determine to some extent the kind of control that is used to execute them in parallel. This is because efficient execution of irregular tasks requires a different coordination model than the execution of regular tasks.
- As described in Section 6.1, the execution of a parallel algorithm is often ruled by internal dependencies. These determine the possible amount of overlap between processes, which is in essence the degree of parallelism. Synchronous activities require common waiting points, while asynchronous activities allow for time-savings due to overlap.

There exists a wealth of control-parallel skeletons in the categories shown above. In this Chapter, we will deal with three special cases:

- *Streams* are commonly used within many Eden skeletons. To deal with them in a safe way is not always easy, as deadlocks can easily occur. We will show a set of methods for dealing with them.
- *Tasks of irregular size* can be executed efficiently by a master-worker skeleton. Some algorithms, however, require extensions of that scheme which we will present here.
- *Long communication distances* within the Eden process tree can be alleviated by introducing dynamic channels cross-connecting processes. We present an implementation of the Hypertree, which represents a binary process tree with additional Hypercube-like cross-connections.

Section 7.2 shows problematic situations that may arise when dealing with streams and discusses methods for solving them. In contrast to the easy to handle regular tasks, irregular tasks are also common and require special treatment: Section 7.3 shows how this can be done with the Workpool scheme. The connections between Eden processes (if dynamic channels are not used) form a tree; in the worst case, it can require many hops to communicate between two processes. Section 7.4 describes the Hypertree skeleton for introducing short-cuts into that tree.

Parts of the chapter are based on our EuroPar paper[173] .

7.2 Dealing with Streams

Infinite lazy lists (as known as streams) can enhance your parallel functional program in many ways. However it is not all roses: We show some caveats which occur in daily stream programming together with techniques on how to avoid them. The

results of this subsection will then be used in the next subsection to define a skeleton based on stream communication.

7.2.1 Introduction

A *stream* is an infinite sequence of values. In a lazy functional language like *Haskell* they can be modelled by lazy lists. Due to non-strict evaluation these can be defined without running the risk of non-termination. Typical well-known functions for stream-generation include: `repeat`, `replicate`, `iterate`, and `cycle`. These share a common basic scheme of circular reference and rely on the combination of non-strict evaluation and sharing (or memoization):

```
ones :: [Int]       repeat :: a -> [a]
ones = 1 : ones     repeat x = xs where xs = x:xs
```

One basic advantage of streams is some kind of *aspect-orientation*: Normally one would write specific functions which combine both the construction of a data structure and a function processing it; now one can separately define the data structure and the processing function:

```
fibs_lt n = flt 1 1 where flt x y | x > n     = []
                                  | otherwise = x : flt y (x+y)
```

↓Separation

```
fibs      = 1 : 1 : zipWith (+) fibs (tail fibs)
fibs_lt n = takeWhile (<n) fibs
```

When dealing with streams we faced a couple of problematic constellations, especially when multiple streams are connected and interdependent. In the following we will show in each subsection a small example exposing a typical problem when using streams and give a solution to the problem or sketch a way showing how to avoid it. For a more complete presentation even quite basic problems are included. A combination of all these solutions will be used in the next section to define a substantial stream-processing skeleton in the parallel Haskell dialect Eden.

7.2.2 Delayed Matching

The following describes a classical client/server relationship. A client executes a stream of tasks and produces a stream of results. The server receives the workers' results and generates new tasks. Note that this mutual dependency can only be opened by including an initial task as the first element of the server's task list.

```
client :: ([Task] -> [Result]) -> [Task] -> [Result]
client execute ts = execute ts

server :: Task -> ([Result] -> [Task]) -> [Result] -> [Task]
server initialTask generate rs = initialTask : generate rs
```

```
tasks   = server init gen results      -- init, gen, ex de-
results = client ex tasks              -- fined elsewhere
```

Now imagine a server checking the client's status first:

```
server :: Task -> ([Result] -> [Task]) -> [Result] -> [Task]
server initialTask generate (r:rs)
  | clientOK r = initialTask : generate (r:rs)
  | otherwise  = error "Client not working!"
```

As matching the results `(r:rs)` (which do not yet exist) occurs *before* generating the first task, this will result in deadlock. Typical solutions include:

- Using *lazy matching* by taking the pattern `~(r:rs)`
- Using *selection functions* `head` and `tail` over the pattern `rs`

Both postpone the matching of the argument against cons from the function entering time to the later expression evaluation time. In general it is useful to defer the matching of arguments to the latest possible point in time.

7.2.3 Incremental Functions

Imagine the client is given the following function for task execution which counts the number of incoming tasks and then creates an average result which is replicated `numTasks` times:

```
ex :: [Task] -> [Result]
ex ts = replicate numTasks averageResult
  where numTasks      = length ts
        averageResult = f ts numTasks
```

The system will again block immediately trying to calculate the length of the task list which does not yet exist completely. The `ex` function is *monolithic*, which means that it will not yield a part of its result until it has completely consumed its arguments. As Hinze[92] puts it:

7.1 Definition (Monolithic)
A list-processing function `f :: Eq b => [a] -> [b]` *is* monolithic *if*

$$\exists n.\forall m, 0 < m < n: \quad \texttt{f . first m} = \texttt{undefined}$$

where

```
first :: Int -> [a] -> [a]
first 0 _      = []
first _ []     = undefined
first n (x:xs) = x : first (n-1) xs
```

Causes for getting monolithic functions include:

- *Problem-inherent dependencies.* The `reverse` of a list cannot be calculated incrementally, although a stream of partial results can be given (see next section).
- *Unnecessary tailrecursion.* Tailrecursion often accumulates the final result in a list parameter and releases it only when a condition has been met. Instead these functions can be transformed to avoiding the accumulator and immediately building the final result which is thus visible even if it is not yet complete.

Now think of an alternative function for task execution like the map-based

```
ex :: [Task] -> [Result]
ex = map f where f t = ...
```

which of course runs smoothly. Such functions are called *incremental*, since for each element of the argument stream an element of the result stream is produced. Again more formal:

7.2 Definition (Incremental)
A list-processing function `f :: Eq b => [a] -> [b]` *is* incremental *if*

$$\forall n: \quad \texttt{first n . f = first n . f . first n}$$

Both definitions can be expressed as Haskell predicates which are suitable for limited testing purposes:

```
test :: Eq b =>
        (a -> b -> Bool) -> ([a] -> [b]) -> [a] -> Int -> Bool
test t f list limit = and vs
  where vs = [ first n (f list) `t` first n (f (first n list))
             | n <- [1..limit] ]
is_incremental = test (==);    is_monolithic = test (/=)
```

Limited because for example `is_incremental` may not terminate for non-incremental functions instead of yielding `False` (semi-decidable).

Between monolithic and incremental functions other functions are located which show a more stagnant character. The following function takes groups of five tasks completely to give back a group of results and is (taken the definitions above) neither monolithic nor incremental:

```
stagnant = s 5 r
  where s :: Int -> ([a] -> [b]) -> [a] -> [b]
        s size reduce xs = reduce group ++ s size reduce rest
          where (group,rest) = splitAt size xs
        r xs = replicate (length xs) (average xs)
```

7.2.4 Partial Result Streams

It is desirable to use only incremental functions on a system of interdependent streams. Unfortunately some functions like `length` generate not streams but values of some base type. To keep streams flowing in an incremental way, one can also produce instead of some base value an incremental stream of result approximations finally containing the full result (see `length'`); additionally, an inherently monolithic function like `reverse` can be turned into a partial result version:

```
length' :: [a] -> [Int]          reverse' :: [a] -> [a]
length' = scanl (\x y -> x+1) 0  reverse' = scanl (flip (:)) []
```

Even approximated results can be useful, in the case of `length'` a function receiving the partial result stream can issue some computation if a given bound is exceeded.

A similar method is given by Okasaki in Section 8.2.1 of his book[161] via introducing state. Reversing a list incrementally could then be done as follows:

```
data Reverse a = Working [a] [a] | Done [a]

reverse' xs                = Working xs []
exec (Working (x:xs) xs') = Working xs (x:xs')
exec (Working []     xs') = Done xs'
```

A series of `exec` applications to the initial `reverse'` application then executes the reversal.

7.2.5 Threads

Very often one needs to *merge* streams into a single one. Three deterministic possibilities to do that are:

```
detDeepM, detFlatPairM, detFlatM, :: [[a]] -> [a]

detDeepM = foldr (++) []

detFlatPairM = foldr1 il
  where il []     ys = ys
        il (x:xs) ys = x : (il ys xs)

detFlatM xss | null xss  = []
             | otherwise = (heads xss) ++ (detFlatM (tails xss))
  where heads []            = []
        heads ([]    :rest) = heads rest
        heads ((x:xs):rest) = x : heads rest
        tails []            = []
        tails ([]:rest)     = tails rest
        tails ((x:[]):rest) = tails rest
        tails ((x:xs):rest) = xs : tails rest
```

`detDeepM` (as known as `concat`) follows each stream to its end and is therefore problematic. `detFlatPairM` merges elementwise but only for pairs of streams. `detFlatM` finally works elementwise on all streams.

But what happens if a stream does not deliver elements, as it is common in a parallel setting? All deterministic versions would block which rises the need for a nondeterministic merge which is predefined in the Glasgow Haskell Compiler:

```
nondetMerge :: [[a]] -> [a]
nondetMerge = unsafePerformIO . nmergeIO
```

`nmergeIO` creates a thread for each substream which nondeterministically appends elements to the merged stream via MVars.

7.2.6 Lazy List Comprehensions

Imagine some values `vals` are to be distributed over a set of streams depending on a list of unsorted stream identifier keys out of a set of all keys. A good way to do that is to use a list comprehension to define each substream by lazily traversing `vals` together with the list of unsorted keys searching for all values matching each substream key.

```
concDistr :: Eq a => [a] -> [a] -> [b] -> [[b]]
concDistr unsortedKeys allKeys vals = result
  where vals'  = zip unsortedKeys vals
        result = [ [v  | (uk,v) <- vals', uk == k]  | k <- allKeys]
```

7.2.7 Accumulations and Limited Access

Searching an infinite stream (with the purpose of eliminating duplicate elements) is a bad idea:

```
nub1, nub2, nub3, nnn :: Eq a => [a] -> [a]
nub1 []    = []
nub1 (x:xs) | elem x xs =     nub1 xs
            | otherwise = x : nub1 xs   -- keep last occurrence
```

Alternatively one can accumulate the first occurrences of each element in the stream and delete forthcoming duplicates or filter out duplicates explicitly. Both versions create space leaks:

```
nub2 = rm []
  where rm acc []     = []
        rm acc (x:xs) | elem x acc =     rm    acc  xs
                      | otherwise  = x : rm (x:acc) xs
nub3 []     = []
nub3 (x:xs) = x : nub3 [e /= x | e <- xs]
```

A better alternative to creating a second list is to lazily refer back to the so far generated list of unique values. This implies that a counter has to be kept which describes the number of unique values found so far; this counter is then used by a version of the `elem` function to avoid searching for not yet existent values in the result list. The resulting function is more space-efficient than its predecessor since no intermediate data structure has to be garbage collected.

```
nnn vs = res
 where res         = n' vs 0
       n' []    _  = []
       n' (x:xs) l | elem' x res l =     n' xs l
                   | otherwise     = x : n' xs (l+1)
       elem' :: Eq a => a -> [a] -> Int -> Bool
       elem' e ~(x:xs) l
        | l == 0    = False
        | otherwise = if e == x then True else elem' e xs (l-1)
       elem' e [] l = False
```

7.2.8 Stream Spreading

When communicating via streams one most often sends nested tuple values. For individual processing these usually have to be split from a stream of tuples into a tuple structure whose elements are streams, for example via using `unzip` repeatedly:

```
spread :: [(a,(b,[c]))] -> ([a], ([b], [[c]]))
spread xs = (as, (bs, css)) where (as, ys)  = unzip xs
                                  (bs, css) = unzip ys
```

Meta-programming with Template Haskell (as shown in Chapter 3) can be used to create the needed `spread` version automatically. Figure 7.1 shows the function `mkSpread`, which creates the needed function version. The auxiliary function `buildP` is not shown as it is almost identical with `buildE`. The initial call is a top-level splice which calls `mkSpread` with a structure describing the tuple nesting. In this case, `spread` will be created for `(a,(b,c))`:

```
$(do let empty       = ListE []
     let tuplestruct = TupE [empty, TupE [empty, empty]]
     let spread_fct  = mkSpread tuplestruct
     return spread_fct)
```

After expansion the splice will be replaced by the needed version of `spread`.

7.3 Dealing with Irregular Task Sizes

Parallelism contained within an algorithm reveals itself in the form of (fully or mostly) independent pieces of work. Determined by the nature of the algorithm and its underlying data structures these pieces (or tasks) can either be

```
mkSpread :: Exp -> [Dec]
mkSpread s = [FunD "spread" clauses]
  where clauses = [Clause ps1 b1 [], Clause ps2 b2 [ValD pat b []]]
        ps1     = [ListP []]
        b1      = NormalB (buildE s (repeat (ListE [])))
        ps2     = [ConP "GHC.Base::" [ps2', VarP "rest"]]
        ps2'    = buildP s [VarP [c] | c <- ['a','b'..]]
        b2      = NormalB (buildE s lists)
        lists   = [AppE (AppE (ConE "GHC.Base::") (VarE [c]))
                   (VarE [c,'s']) | c <- ['a','b'..]]
        pat     = buildP s [VarP [c,'s'] | c <- ['a','b'..]]
        b       = NormalB (AppE (VarE "spread") (VarE "rest"))

buildE :: Exp -> [Exp] -> Exp;   buildP :: Exp -> [Pat] -> Pat
buildE (TupE vs) ls = TupE rs
  where (rs, _)                        = trav vs ls
        trav ((ListE []):rest) (l:ls) = (l : r, ls')
          where (r, ls')   = trav rest ls
        trav ((TupE ws):rest) ls      = ((TupE rec):r,ls'')
          where (rec, ls') = trav ws ls
                (r, ls'')  = trav rest ls'
        trav [] ls                    = ([], ls)
```

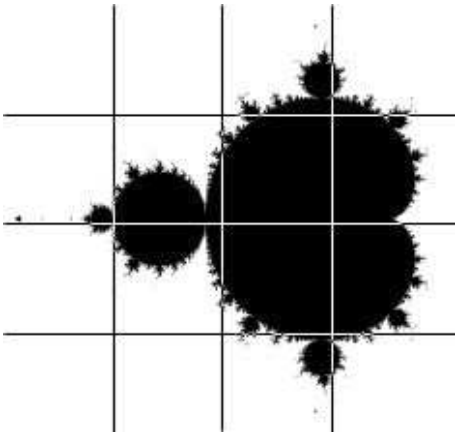
Figure 7.1: Template Haskell generation of `spread` (`buildP` omitted)

- *regular*, which means that the amount of processing time is approximately the same for each piece of work (assuming a homogeneous set of processing nodes)

- or *irregular*, which means that the pieces differ considerably. As their true complexity can often hardly be estimated, it is also not easy to combine them to form a collection of similarly sized tasks.

Regular tasks are usually organised in a collection and lend themselves to being processed with data-parallel skeletons. These are perfectly suited for distributing equally-sized tasks over a network of uniform computing nodes to easily achieve a predictable load-balance.

On the other hand, uneven task sizes arise naturally from many problems and are often an obstacle to static parallelisation. Even if data structures are divided into portions of equal sizes, this does not imply that the amounts of work induced by those portions are also equal. Take for example the parallel calculation of the Mandelbrot set [119] . This set is computed over a range of complex coordinates by iterating a function until a certain approximation has been reached; the number of iterations needed determines the color of the corresponding pixel. Some pixels need only a few iterations, while others need many more. As computation-intensive pixels are grouped together, a regular partitioning leads to irregular granularity:



In this graphic, white pixels represent short and black pixels represent long computation times. If these partitions are equally distributed over the parallel nodes, a serious load-imbalance would arise. (But if portions have to be laid out evenly in a static manner, a partitioning into portions of varying sizes would be necessary.)

Such irregular tasks, on the other hand, cannot be handled that easily. Within parallel dialects of Haskell the classical static distribution schemes (like parallel *map*) can hardly establish load-balance given unevenly sized tasks; therefore dynamic task distribution schemes are used. The well-known *workpool* scheme (also known as farm, master-worker, or client-server)[119] is mostly used to compensate for such irregularly sized tasks: A master administrates a statically fixed task pool out of which tasks are gradually assigned to currently idle workers, leading to a balanced workload. Such a scheme is often expressed as a high-level code template, known as a *skeleton*[45, 176] .

In the following subsections we will present a series of Eden workpool skeletons (see also our previous work[119, 173]), each tailored for a different purpose. Together they provide means for dealing with irregular task sizes in many facets.

7.3.1 Basic Workpool

Within the basic workpool[119] , a master process keeps a pool of tasks which are distributed to a set of worker processes on request. When a worker receives a task, it solves it and sends back the result which is interpreted as a request for new work. This way each worker is busy most of the time and load balance is kept as tasks are assigned depending on the current work distribution.

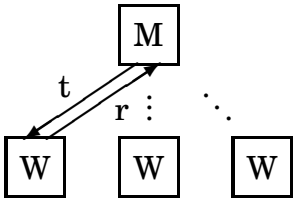


Figure 7.2 shows the code for the basic workpool. The skeleton takes as arguments:

- the number of worker process that should be created (having n nodes, one would ideally create $n-1$ workers and leave one node for the master process)

```
wpool :: (Trans t, Trans r) =>
          Int ->                        -- #workers
          Int ->                        -- prefetch size
          (t -> r) ->                   -- worker function
          [t] -> [r]                    -- tasks, results
wpool np prefetch f tasks = results where
  fromWorkers        = eagerInst1 workerProcs toWorkers
  workerProcs        = [process (zip [n,n..] . map f) | n<-[1..np]]

  toWorkers          = distribute tasks requests
  (newReqs, results) = (unzip . merge) fromWorkers

  requests           = initialReqs ++ newReqs
  initialReqs        = concat (replicate prefetch [1..np])

  distribute :: [t] -> [Int] -> [[t]]
  distribute tasks reqs = [taskList reqs tasks n | n<-[1..np]]
    where taskList (r:rs) (t:ts) pe | pe == r    = t:(taskList rs ts pe)
                                    | otherwise  =    taskList rs ts pe
          taskList _      _      _  = []
```

Figure 7.2: Code for Basic Workpool

- a prefetch number for ordering tasks in advance (prefetch n means, that for each worker n requests for work are preassumed)
- a function which each worker executes and the task set.

The whole data flow of the skeleton is modelled as a set of interdependent streams; within this basic workpool, stream handling is not yet difficult. At first, all worker processes are created via the generic function `eagerInst1`. This function takes a structure of process abstractions and another structure of arguments and eagerly creates all processes. The whole result is a structure of results:

```
eagerInst1 :: (Abs1 t, Trans a, Trans b) =>
              t (Process a b) -> t a -> t b
eagerInst1 ps xs = let insts = gZipWith1 (createProcess) ps xs
                   in tl1 insts
```

`toWorkers` is a list of argument streams (one for each worker) which is created by distributing the tasks of the task pool depending on the current `requests` for work. The requests for work are defined by a set of initial requests determined by the prefetch parameter, together with the new requests which are derived from the results sent by the workers. The worker processes themselves operate on a stream of tasks by mapping their worker function and a marking function onto the task stream. The marking function just pairs the result with the worker number; this information is later used as a new request.

There are two peculiarities influencing the performance of the workpool which are connected to the way communication is handled by the Eden implementation. Messages are sent via one of the typical message-passing libraries PVM [160] or MPI [152, 153] , which is quite costly and not worthwhile if the message is smaller than a certain limit. In contrast, programming in a parallel function language like Eden can easily tempt the programmer to define communication 'in the small' by encouraging the use of stream processing of small elements. This can lead to small data structures being transmitted via costly message sending which has to be avoided. Two situations can occur:

- Task sizes are allowed to vary, as the dynamic assignment to workers will compensate their different sizes. Too small tasks on the other hand can harm the performance, as they may be delivered by costly messages. Therefore task sizes should be allowed to vary only above a certain level.
- As with every parallel program it is important to quickly get every parallel node working. For the workpool this means, that in the beginning every worker has to start working as quick as possible. The prefetch parameter shown above will speed up the distribution of work in the starting phase of the workpool, as the master will send a supply of tasks to each worker independent on receiving the corresponding requests. In the starting phase this eliminates the typical pause consisting of the pause between sending a result (i.e. a request) and receiving a new task.

  Another improvement is the addition of an initial task to the worker function in the style of the direct mapping skeleton of Chapter 6. Then each worker process is created carrying its first task already in itself:

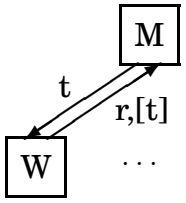
```
wpool np prefetch f tasks = results where
  ...
  workerProcs        = [ process (\ts -> (zip [n,n..] . map f)
                           ((ts1!!n):ts)) | n <- [1..np] ]

  (ts1, ts2)         = splitAt np tasks
  toWorkers          = distribute ts2 requests
  ...
```

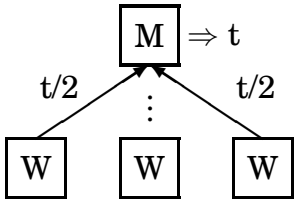
7.3.2 Dynamic Workpool

In contrast to the workpool just shown, some applications expose their full task set only successively as the computation proceeds and need therefore a more general workpool skeleton whose worker processes are allowed to generate new tasks dynamically. Then a task will not only produce a result, but possibly also a set of new tasks for the global task pool. This introduces the problem of termination detection, which was not a problem before since a statically fixed task number makes it easy to determine termination: Given n tasks, the master can terminate the workpool if n results have been received. Now special care has to be taken to account for

a dynamically growing and shrinking task pool. Emptiness of the the task pool does no longer mean that there is no more work to do, since new work may still be created by active workers.



To make things even more complicated, dynamically created tasks may be incomplete (e. g. due to limited local data) and need to be combined with other incomplete tasks before submission to a worker. Therefore means have to be provided for traversing and transforming the task pool on the fly. But since the task pool is often modelled as a lazy list and woven into a network of interdependent streams, one has to be extra careful during a transformation. When combining partial tasks, deadlocks can easily occur by searching for not yet existent partial partner tasks; additionally, all usual techniques (like delayed pattern matching and incremental functions shown in the previous Section) when dealing with lazy lists have to be considered.



Therefore we extend the basic workpool scheme by two new features:

- *Dynamic task generation.* When a worker processes a task new tasks may arise. These will be sent back to the master and appended to the global task pool, preserving task order.
- *Task pool transformation.* Sometimes it is helpful to be able to process and transform the task pool. A given transformation function `tt` will be applied to the task pool to combine incomplete tasks and replace them by complete ones, possibly changing task order.

The resulting basic interaction scheme is shown in Figure 7.3. All connections shown are stream connections; the thick pointers touching the worker processes Ⓦ are interprocess connections while all others reside within the master process. The diagram directly reflects the informal workpool description given above together with our two extensions. After a task pool transformation by the function `tt`, tasks are distributed (depending on the requests at hand) over the idle worker processes. The list of result streams is then merged and unzipped via `spread` to yield a result stream, a list of streams containing new tasks, and a list of new requests. Figure 7.4 shows the full code for the extended workpool. Parameters are: The number of processors available, the number of advance requests for each worker, the worker function, the transformation function for the task pool, and finally a set of initial tasks. At first, the workpool demands the first cons of the list

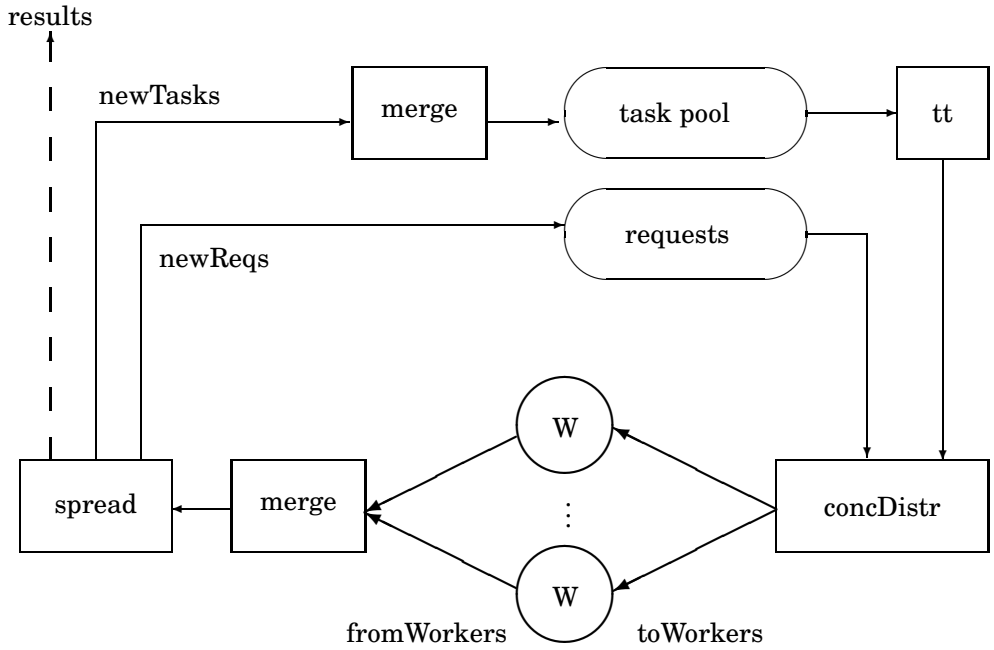


Figure 7.3: Stream interconnections of workpool (seen from master process)

of worker processes via `touch` to trigger their creation using the predefined parallel zip function `eagerInst1`, then the `results` are given back. `concDistr` is taken from Section 7.2.6, and `mkSpread` from Section 7.2.8. The main skeleton body is divided into two parts:

The *stream part* directly transfers Figure 7.3 into a set of interdependent stream-processing functions. A set of `workerProcs` is created, which apply the worker function `f` to their input and attach their id number to the result as a request for new work. Their input `toWorkers` is a list of streams, each of which contains `tasks` for the corresponding worker according to its `requests`. The `initialRequests` are built based on an interleaved sequence (each of size `prefetch`) of worker numbers and provides an initial supply of tasks for each worker. The worker's outputs are merged to a single `workerstream` which is `spread` to yield a tuple of streams instead of a stream of tuples. New requests are appended to the list of pending requests while new tasks are added to the task pool which gets transformed by `tt`. Now one could finish the transfer by extracting the `results` out of `spread` via `x`. Wrong, because this would result in non-termination since after processing all tasks the master would wait forever for further worker messages containing new tasks.

Therefore the *state part* has been introduced to care for termination detection and result accumulation. The `terminate` function carries a state consisting of a set of incomplete tasks, the number of complete tasks in the task pool, the accumulated results, and the number of accumulated results. In addition to the continuous evaluations in the stream part, `terminate` traverses `workerstream` a second time in a stepwise fashion. For every answer from a worker process, `terminate` will run `tt` on the incomplete tasks extended by the received new tasks and update its `t` counter accordingly. The result counter `r` is incremented by 1, as every answer delivers exactly one result. If then the new counters `t'` and `r'` are equal, which means that for every complete task issued to the task pool a result has been received, the

```
wpool :: (Trans t, Trans r) =>
         Int -> Int -> ([t] -> [(r, [t])]) ->
         (([t],[t],[t],Int) -> ([t],[t],[t],Int)) ->
         [t] -> [r]
wpool np prefetch f tt initialTasks =
  (touch fromWorkers) `seq` results
  where touch []    = ()       -- Demand first constructor to
        touch (_:_) = ()       -- initiate worker creation

        -- 1) Streams and communication --------------------------
        fromWorkers = eagerInst1 workerProcs toWorkers
        workerProcs = [process (zip [n,n..] . f) | n<-[1..np]]

        toWorkers   = concDistr requests [1..np] tasks
        requests    = initialReqs ++ newReqs
        initialReqs = concat (replicate prefetch [1..np])

        taskpool                 = initialTasks ++ (merge newTasks)
        (_, _, tasks, _)         = tt (taskpool, [], [], 0)
        workerstream             = merge fromWorkers
        (newReqs, (x, newTasks)) = spread workerstream

        -- 2) State and termination ------------------------------
        ([], _, results, _) = terminate
                              ([], length initialTasks, [], 0)
                              workerstream

        terminate (is,t,rs,r) ( (_,(res,ntasks)) :ws)
          | t' >  r' = terminate (is', t', res:rs, r') ws
          | t' == r' = (is', t', reverse (res:rs), r')
          | t' <  r' = error ("Will never happen.")
          where ([], is', _, n) = tt (is++ntasks, [], [], 0)
                t'              = t + n
                r'              = r + 1
        terminate _           []
          = error "Workerstream empty!"

-- TH splice creates: spread :: [(a,(b,[c]))] -> ([a], ([b], [[c]]))
$(do let empty      = ListE []
     let structure  = TupE [empty, TupE [empty, empty]]
     let spread_fct = mkSpread structure
     return spread_fct)
```

Figure 7.4: Workpool with dynamic task generation and task pool transformation

workpool terminates giving back the reversed list of results. If, on the other hand, `t' > r'`, then the remaining incomplete tasks together with the new counters and the result list will be used for a tail-recursive call to `terminate`. The remaining case `t' < r'` can never happen since every step will yield only one result.

When constructing a proper task pool transformation function `tt` for the workpool one has to be careful because:

- In the stream part `tt` is applied once to a stream of tasks while in the state part it is applied many times to a finite task pool. It has to behave correctly in both situations.
- As interdependent task and result streams are used it is necessary to produce as much output as possible with as few inputs as possible. Therefore delayed matching (via the lazy matching operator `~` or selection functions `head` and `tail`) and the earliest possible production of results should be used.
- Transformation often means combination or comparison which implies searching the task pool. As the task pool is potentially infinite one runs the risk of searching for (and then blocking on) not yet existent tasks.

As `tt` will often in some way have to combine incomplete tasks to complete ones, we present in Figure 7.5 a predefined function `ttransform` for doing this while taking some care of the aforementioned dangers. One has to provide only two arguments to `ttransform` to get a full version of `tt`: Firstly, a predicate `cp`, which

```
ttransform,ttransform2 :: (t -> Bool) ->              -- complete, cp
                          ([t] -> ([t],[t],Int)) ->   -- combine,  co
                          ([t], [t], [t], Int) -> ([t], [t], [t], Int)
ttransform cp co old@(tasks, incomplete, complete, n)       -- Step 1
  = if (not (null incomplete))
    then let (ct,it,d) = co incomplete
         in if (not (null ct))
            then ttransform  cp co (tasks,it,ct++complete,n+d)
            else ttransform2 cp co old
    else ttransform2 cp co old

ttransform2 cp co (t:ts, incomplete, complete, n)           -- Step 2
  | cp t      = (tts1,iis1,  t:ccs1, d1+1)
  | otherwise = (tts2,iis2,ct++ccs2, d2+d)
    where (tts1,iis1,ccs1,d1) = ttransform cp co
                                (ts, incomplete, complete, n)
          (ct,it,d)           = co (t:incomplete)
          (tts2,iis2,ccs2,d2) = ttransform cp co (ts, it, complete, n)
ttransform2 cp co ([],incomplete, complete, n) = ([], it, ct, n+d)
  where (ct,it,d) = co incomplete
```

Figure 7.5: Higher-order function `ttransform` for task pool transformation

checks whether a given task is *complete* or not. Secondly, a function `co` which takes a set of mixed complete and incomplete tasks and tries to *combine* as many incomplete tasks as possible. Its results are the already complete tasks together with the newly completed tasks, the currently not combinable incomplete tasks, and the number of newly generated complete tasks.

To avoid the above mentioned danger of blocking when trying to find partners for incomplete tasks we will make only a single traversal over the task list and use an accumulator to carry not yet combined tasks with us. For that purpose `ttransform` carries a state argument consisting of the remaining task stream, the accumulator, a stream of complete tasks (its result), and the number of new complete tasks (needed to correct termination detection counters). `ttransform` is divided in two steps: The first step postpones any matching on the input task stream and tries to combine incomplete tasks inside the accumulator as long as possible. Only if that fails, it matches the first task of the task stream and acts depending on its completeness. Complete tasks are immediately passed to the output stream while incomplete ones are tried to be completed. By considering data dependencies the user has to make sure that enough complete or completable tasks are generated in the right order by his application.

7.3.3 Nested Workpool

A growing number of workers or tasks induces heavy traffic at the master process which then apparently quickly becomes a bottleneck for the whole workpool scheme; additionally the dynamic task pool transformation means even more work for the master process which worsens the bottleneck it already presents. This can be alleviated by having more independent workers which manage a buffer of tasks for themselves. In other words: We will replace each worker by another workpool for local task distribution to distribute the administrative load.

Figure 7.6 shows the code for such a *nested workpool* with an even arbitrary nesting depth ≥ 1. For depth 1 the previously defined workpool is returned. The depth is controlled by the (equal) length of the first three argument lists which contain the number of workers (or submasters respectively), the prefetch, and the task transformation function for each level of the workpool tree. The nesting itself works by folding the zipped arguments for each level with the `wpool` function. The

```
wpN :: (Trans t, Trans r) =>
  [Int] -> [Int] ->                                -- #workers, prefetches
  [(([t],[t],[t],Int) -> ([t],[t],[t],Int))] -> -- transformations
  ([t] -> [(r,[t])]) ->                            -- worker function
  [t] -> [r]                                       -- tasks, results
wpN ns pfs tts f initTasks = results where
  (results,_)      = unzip ((foldr fld f (zip3 ns pfs tts)) initTasks)
  fld (n,pf,tt) wf = \ts -> zip (wpool n pf wf tt ts) (repeat [])
```

Figure 7.6: Nested workpool

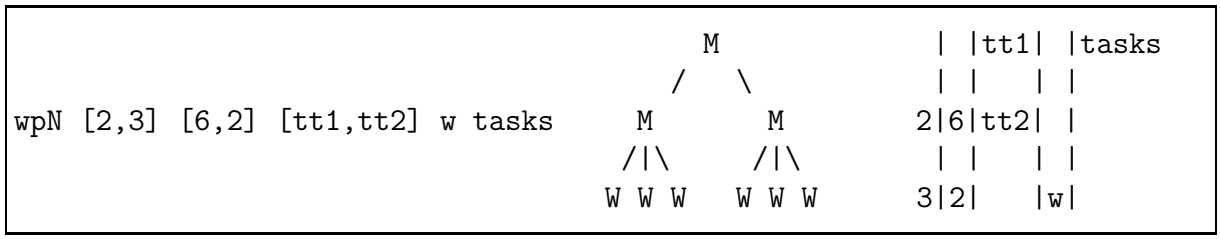


Figure 7.7: Two-level example `wpN` call with process tree and argument distribution

worker function `f` is used to close the workpool tree with a set of worker leafs. Note the use of `repeat`: No submaster will migrate tasks to masters above him, therefore newly created tasks will only be sent by the worker leafs to their respective master processes. For better understanding Figure 7.7 shows an example call of the nested workpool together with the resulting process tree; additionally for each argument it is shown to which level it applies. The termination detection of `wpool` fits smoothly into this nesting.

7.3.4 Dynamic Workpool with Stateful Master

Due to redundancies workers within a dynamic workpool may generate new tasks which are equal or (in comparison to others) even outdated and unneeded. This is the case for example for branch-and-bound algorithms: A tree of solutions is traversed searching for a maximal solution. Using the currently maximal solution whole parts of the search tree can be pruned and left out. Such methods require a local state, in which the current progress of the algorithm can be stored. The dy-

```
wpoolS :: ...
          (t -> r) ->                           -- worker function
          (state -> [r] -> (state, [t])) ->     -- tt function
          state ->                              -- initial state
          [t] -> state                          -- initial tasks, result
wpoolS np prefetch function tt s1 tasks =
  (touch fromWorkers) `seq` s2
  where ...
        (s2, newtasks) = tt s1 results
        ...
```

Figure 7.8: Dynamic workpool with stateful master

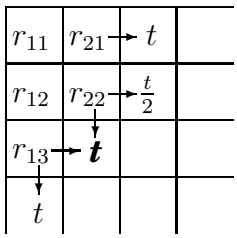
namic workpool can easily be adapted to carry state (see Figure 7.8). This is done by cutting away the termination detection since now the master will trigger termination based on its own state. The task transformation function now traverses the result stream and updates its initial state `s1` depending on these results. Furthermore, it uses these results to generate the new tasks. This of course shifts work from the workers to the master. As soon as the final state `s2` has been generated the workpool terminates.

7.3.5 Example

We have used the extended workpool of Section 7.3.2 to parallelise the alignment of DNA sequences via the linear Needleman-Wunsch [155] algorithm. Although not being very efficient, the algorithm serves as a good example for wavefront parallelism [9] : Within a matrix structure the algorithm exhibits diagonal wavefront dependencies (see Figure 7.9) which can be expressed as tasks for execution via the extended workpool. More specifically: Each block depends on its two left and upper neighbours in the matrix. Therefore each result produces incomplete tasks for its right and lower (not yet computed) neighbours. Two of these incomplete tasks will then be combined in the task pool to form a new complete task. Only elements of the first row and the first column can be computed given only one of their respective neighbours.

Figure 7.10 shows on the left the relative speedup of the parallel sequence alignment algorithm using the extended workpool. All measurements were taken on a cluster of nine Linux PCs connected via 100 Mbit ethernet. The PCs are not completely identical, but this is compensated by the dynamic task distribution of the workpool. Sequences of length 10.000 have been tested with a varying block partitioning. The figure shows that a medium task granularity (block size 500) has paid off the most in our experiments. Larger tasks result in task shortage, while smaller tasks induce too much administrative overhead due to their large number. The nested workpool cannot be used to reduce that overhead, since tasks of different subworkpools would have to be combined. We are aware that our unoptimised implementation of a suboptimal algorithm is slower than modern imperative alignment solutions; it nevertheless serves as a good example of wavefront parallelism for our workpool.

Figure 7.10 shows on the right an activity diagram for the execution of the parallel sequence alignment on nine processors (length 10.000, block size 500). Each row represents the activity of one processor during execution, starting on the left and ending on the right at around 45 seconds. White areas represent phases of inactivity or blocking on not yet available data (combined for better visibility), while black areas represent active computation or communication. The lowest row (processor 1) contains the master process which shows constant activity in distributing and combining tasks. The remaining rows show the activity of the worker processes. These are evenly loaded with tasks. Both start and end phase of the computation show clearly the growing and shrinking task availability induced by the diagonal wavefront traversal of the matrix described in Figure 7.9.



Creation of task t_{23}:
1) t_{13} ends, yields r_{13}, releases t_{14} and $(\frac{t}{2})_{23}$ into the task pool
2) t_{22} ends, yields r_{22}, releases $(\frac{t}{2})_{23}$ and $(\frac{t}{2})_{32}$ into the task pool
3) `tt` creates t_{23} in the task pool
4) t_{23} will later release $(\frac{t}{2})_{24}$ and $(\frac{t}{2})_{33}$ into the task pool

Figure 7.9: Task creation for PSA ($\frac{t}{2}$ represents an incomplete task)

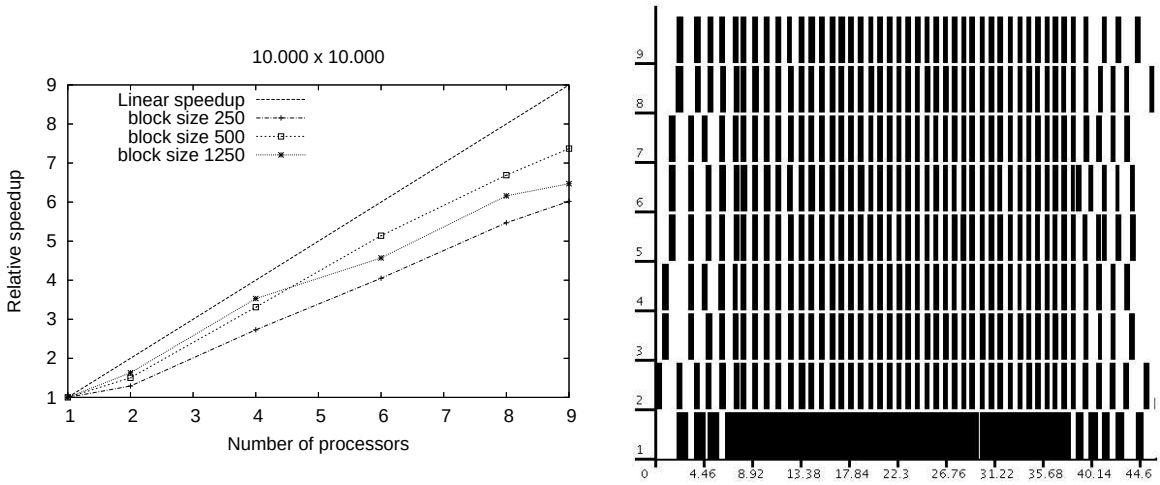


Figure 7.10: Relative speedups and activity diagram (length 10.000, 9 nodes)

7.4 Dealing with Long Communication Distances

On the first sight, Eden process trees have to follow the tree-like topology of functional expressions. Due to the restrictions and disadvantages of this topology in a parallel setting, it is possible to break up this topology to form more flexible ones (encoded as skeletons) using dynamic channels. In this section we introduce cross connections into the Eden process tree forming a *hypertree*.

7.4.1 The Eden Process Tree

In a functional program, expressions and functions are nested and form, by and large, a rose tree. Due to laziness circles can be introduced, which enable the programmer to concisely define infinite data structures (like the Hamming numbers shown in Subsection 5.3.2). The same is true for Eden, with the difference that process applications are inserted into the tree. Therefore the process topology normally created also resembles a rose tree with communication flowing up and down the tree. This is satisfactory as long as not too much communication between processes of different branches occurs. If this happens (see Figure 7.11), the long distances with many process-to-process hops will make communication expensive. The root process will then additionally be a bottleneck, which we have already seen when dealing with the workpool master in the previous section.

If processes are created carelessly, the resulting communication topology can even be totally different to what was planned in the first place. As has been shown by Berthold and Loogen[16] , an intended ring of processes can degenerate to a tree with all communication flowing through the root process. This situation can be solved by using dynamic channels to cut short communication ways by directly linking two processes. Using these, the standard communication topologies ring, torus, and hypercube have already been implemented.

Looking again at the conventional Eden process tree, it is clear that extensive communication in an unforeseen nature between remote nodes will be quite expen-

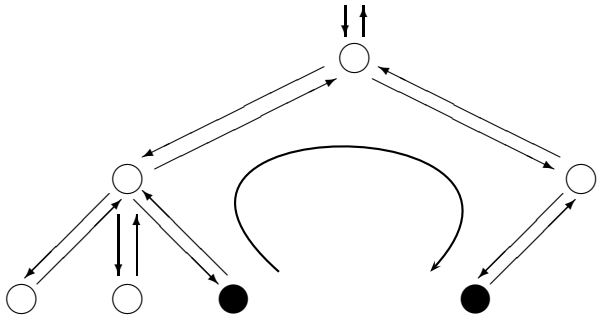


Figure 7.11: Conventional Eden process tree with long communication (four hops)

sive. As an alternative to using a hypercube, we will now look at another way to introduce short-cuts into the tree while using less additional interconnections than the hypercube does.

7.4.2 The Hypertree

The *Hypertree*[73, 58] has been designed to combine features of the hypercube and the binary tree. It is basically a binary tree whose nodes are numbered level-wise such that the children of node n are nodes $2n$ and $2n + 1$; the parent of node n is then node $n \; div \; 2$. Additionally, level-wise interconnections (see Figure 7.13) are added between selected nodes. However, not all nodes on each level are connected; instead, nodes which get a direct connection are chosen depending on their distance in the tree.

Interconnections are added incrementally and level-wise. Firstly, for each level, every pair of nodes whose number in binary representation differs in only one bit is selected. Then for each pair the number of hops needed to reach each other is counted; when determining the distance, the new interconnections above the current level can already be used. For the largest distance, all pairs differing in that bit are then connected via an additional connection. By doing so, for each level the largest communication distance is reduced to one.

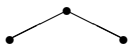
As an example, we will look at level 1. Level 0 contains only the root node and is therefore skipped. In the following, we will count bits starting from the most significant bit (the first 1 occurring when searching from the left). For example, the node number 5 in binary representation is referenced bitwise as follows:

	Bit 0	Bit 1	Bit 2
0 0	1	0	1

Figure 7.13 also shows the bit representations for four levels with a line marking the most significant bit within each representation. The two nodes of level 1 then differ in bit 1. Their distance is 2 and as this is the only pair a direct connection is constructed:

Level↓, MSB→	1	2	3	4	5	6	7	
001	**2**							→ bit 1
010	**3**	2						→ bit 1
011	3	**4**	2					→ bit 2
100	**5**	3	4	2				→ bit 1
101	3	5	**6**	4	2			→ bit 3
110	5	**7**	3	6	4	2		→ bit 2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮

Figure 7.12: Determination of address bits for node connection



Level 2 is different: For bit 1 two node pairs occur with communication distance 3. Two other node pairs (for bit 2) only have a distance of only 2, therefore the first two pairs (bit 1) are connected:

This is continued for the following levels. Figure 7.12 shows the communication distances for each bit up to level 6. The longest distance within each level is chosen for a short-cut and printed in bold style. The resulting bit numbers which determine node pairs connected by these short-cuts are shown in the last column. As shown by Goodman and Séquin[73] , the bit positions can also be calculated via

$$b = \frac{levelnumber}{2^{z+1}} + \frac{1}{2}$$

where z is the number of consecutive trailing zeros in the level number in binary representation. If the nodes defined by this method are connected, the hypertree of Figure 7.13 is the result: A binary tree, within which the longest communication distances are shortened by adding accurately chosen interconnections.

We will now implement the Hypertree as an Eden skeleton in a stepwise fashion.

7.4.3 A Hypertree Skeleton for Eden

Figure 7.14 shows the basic definitions used within the Eden hypertree skeleton. Each node is identified via its code, which is represented as an `Int`. The skeleton will finally be modelled as a tree of processes connected by streams; via these streams, messages `Msg a b s` will be routed. A message can be one of four types:

- `Output` contains a result which is always routed to the father node and included into the result stream

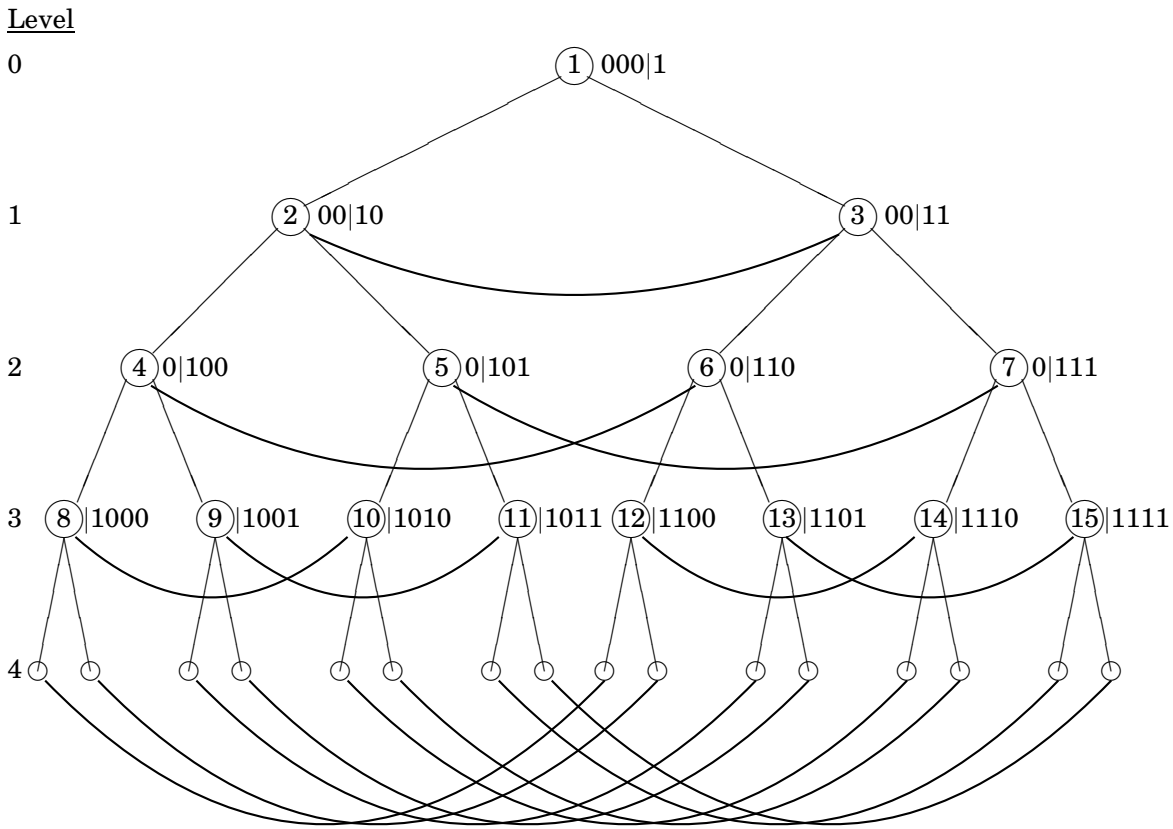


Figure 7.13: A hypertree with binary node encodings (MSB marked)

- `Payload` is used for transmitting a value from one node to another
- `Ref` contains a reference to a dynamic channel and is used for building up cross connections in the starting phase of the tree
- `State` contains a state generated by a node and is also routed always to the father node which updates its internal state on arrival

The `levelnumber` function computes for a node the level in the tree to which it belongs. Functions for calculating the parent and the children of a node are based on the standard level-wise ordering. More interesting is the `partner` function which, for a given node, calculates its partner node to which it is connected via a hyper connection. Every node has a partner except the father node. As we have seen in Figure 7.12, the level number of a node can be used to calculate its partner node address. To avoid constant recalculation of partner bits, the first 20 bits have been precalculated; this is sufficient for 2^{20} leaf nodes or $2^{20+1} - 1$ nodes in total, which is by far enough for a parallel system. Given a node address, the function first determines its level, then selects the bit to be changed out of the offsets list, determines the most significant bit of the node address, and finally applies the offset to that position to find the bit which has to be complemented for finding the node's partner.

Messages are routed within the hypertree via a central function `route` (see Figure 7.15). It takes the addresses of the current and the destination node and cal-

```
type Code = Int
father    = 1

data Msg a b s =
    Output  b                                  -- Output, route to father
  | Payload Code Code a                        -- Payload Src -> Dst
  | Ref     Code Code (ChanName [Msg a b s])   -- DC      Src -> Dst
  | State s                                    -- State to father

levelnumber :: Code -> Int
levelnumber code = find ((bitSize code)-1) code
  where find counter c | testBit c counter = counter
                       | otherwise         = find (counter-1) c

leftChild, rightChild, parent :: Code -> Code
leftChild  code = shiftL code 1            -- x -> 2x
rightChild code = setBit (shiftL code 1) 0 -- x -> 2x + 1
parent     code = shiftR code 1            -- x -> x div 2

partner :: Code -> Code
partner code = code' where
  code'     = complementBit code bit'
  offsets   = [1,1,2,1,3,2,4,1,5,3,6,2,7,4,8,1,9,5,10,3] -- height 20
  offset    = offsets!!((levelnumber code)-1)
  bit'      = msb code (bitSize code) - offset
  msb code cnt | testBit code cnt == True = cnt
               | otherwise                = msb code (cnt-1)
```

Figure 7.14: Binary coding of nodes and basic definitions

culates the next neighbouring node to which the message is forwarded. This way the message travels node-wise to its destination while each node determines the next node the message will be sent to. Messages can be routed in different directions captured by the datatype `Route`: up to the parent, left or right to the children, or sideways via a hyper connection. Not every node offers every direction, as for example a leaf node has no children. Additionally we have to be able to deal with an incompletely constructed hypertree, as the calculations within the nodes are allowed to start while the tree is still unfolding itself; this means, that in the beginning a `Hyper` connection may not be available but is added later if the connection has been established. Therefore the `route` function is also supplied with a set of possible communication ways it may use at the moment the routing decision is made.

The routing itself is driven by the level numbers of the source and the destination nodes. If the destination node is located on a lower level (towards the leafs), the subtree of the current node has to be taken into account: If the destination node is part of that subtree, then it can simply be routed downwards to one of its chil-

```
data Route = Up        -- 0: father
           | Hyper     -- 1: hyperspace
           | Lefty     -- 2: left child
           | Righty    -- 3: right child

route :: Code ->     -- Current node
         Code ->     -- Destination node
         [Route] ->  -- Possible ways (always: Up)
         Route       -- Next stop
route me dst ways
  | dst == father = Up
  | me == dst     = error "Self removed, should be intercepted earlier!"
  | meL < dstL    = if (dst `inSubtreeOf` me)
                    then if (dst `inSubtreeOf` (leftChild me))
                         then Lefty
                         else Righty
                    else if (elem Hyper ways && useful)
                         then Hyper
                         else Up
  | meL >= dstL   = if (elem Hyper ways && useful)
                    then Hyper
                    else Up
  | otherwise     = Up
  where meL    = levelnumber me
        dstL   = levelnumber dst
        pme    = partner me
        useful = hdist pme dst < hdist me dst

hdist :: Code -> Code -> Int
hdist c1 c2 = sum [1 | b <- [0..(bitSize c1)], testBit c b]
  where c = xor c1 c2

inSubtreeOf :: Code -> Code -> Bool
inSubtreeOf search root
  | search == root = True
  | search <  root = False
  | search >  root = if (even search)
                     then inSubtreeOf (search     `div` 2) root
                     else inSubtreeOf ((search-1) `div` 2) root
```

Figure 7.15: Routing within the hypertree

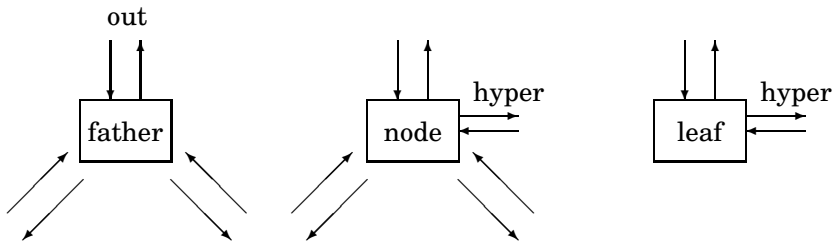


Figure 7.16: Data flow in Eden hypertree (father, node, and leaf)

dren. If it is not part of that subtree, the message has to change the subtree. This is done first via checking if using a hyper connection is possible and useful. If it is not, then the message is routed upwards. As shown by Goodman and Séquin[73] , a hyper connection is useful if the Hamming distance between the current and the destination node is bigger than the Hamming distance between the node reached via the hyper connection and the destination node. But a hyper connection is not always the best choice, even if the Hamming distance is reduced; the sequence of checks in the `route` function is needed to ensure proper routing, which is still not optimal. If the destination node lies on the same or a higher level (towards the father node) than the current node, it is checked whether a hyper connection is useful. A message to the father node is always routed upwards.

We will now start to construct the skeleton itself. Figure 7.16 shows the all three kinds of nodes contained in the tree together with their final stream connections. The skeleton itself consists of two functions, the main function `hypertree` and the auxiliary function `hyperproc`, which is an Eden process abstraction. The main part of `hypertree` is shown in Figure 7.17. Its arguments consist of

- the number of levels,
- the worker function executed by each node (except the father node) which essentially processes a stream of messages
- four arguments for keeping state,
- an input message stream and a result stream.

Its result is twofold: First, a sequence of side effects is touched. These consist of the creation of the left and the right child process via the second function `hyperproc` we will show later. Second, it returns its output `outExtern`. In total the father will act as a message distributor. It merges the incoming streams from outside and from its two children and routes them via `distr` either to the outside or to its children. `distr` is then the central steering function of `hypertree`. It keeps a state which is updated if

- using `updateO` when an output message arrives
- using `updateC` when a state message arrives.

```
hypertree :: (Trans a, Trans b, Trans s2) =>
  Int ->                                        -- Tree height (height-1..0)
  (Code -> [Msg a b s2] -> [Msg a b s2]) -> -- Node function (not father)
  s ->                                          -- / Initial state
  (s -> s2 -> s) ->                             -- | Update state (child)
  (s -> b -> s) ->                              -- | Update state (output)
  (s -> Bool) ->                                -- \ Terminate?
  [Msg a b s2] ->                               -- Input message stream
  [b]                                           -- Output stream
hypertree height nf init updateC updateO finished inExtern
  | height >= 2 = side_eff `seq` outExtern
  | otherwise   = error "Height too small! (<2)"
  where
   side_eff = inLeft' `seq` inRight'
   inLeft'  = cP (hyperproc (height-2) (leftChild  father) nf) outLeft
   inRight' = cP (hyperproc (height-2) (rightChild father) nf) outRight
   cP       = createProcess
   inLeft   = deLift inLeft'
   inRight  = deLift inRight'
   inAll    = inExtern ++ (merge [inLeft, inRight])
   (outExtern, outLeft, outRight) = distr init inAll

   distr st []                    = ([], [], [])
   distr st (Output b      : is) | finished st' = ([b],  [], [])
                                 | otherwise    = (b:oe, ol, or)
     where (oe, ol, or) = distr st' is
           st'          = updateO st b
   distr st (p@(Payload s d v) : is) = if d == me
     then error "Payload to father!"
     else case way of Lefty  -> (oe, p:ol, or)
                      Righty -> (oe, ol, p:or)
     where (oe, ol, or) = distr st is
           way          = route me d ways
   distr st (r@(Ref s d c) : is) = if d == me
     then error "Ref to father!"
     else case way of Lefty  -> (oe, r:ol, or)
                      Righty -> (oe, ol, r:or)
     where (oe, ol, or) = distr st is
           way          = route me d ways
   distr st (State s : is) | finished st' = ([], [], [])
                           | otherwise    = distr st' is
     where st' = updateC st s
   ways = [Lefty,Righty]
   me   = father
```

Figure 7.17: The Hypertree skeleton

The state triggers termination: If the `finished` function yields `True`, all output streams are closed which leads to the successive shutdown of the hypertree.

The `hyperproc` function (see Figure 7.18) is the process abstraction which is run for every node except the father which only deals with routing and state keeping. Only one process abstraction is used to cover both internal tree nodes and tree leafs; `hyperproc` is therefore divided in two parts. Definitions common to both are shared (see Figure 7.19) and also part of the `where` block. We will leave out the details for the leaf implementation, as they are similar to the node implementation. The process abstraction again first touches the side effects which trigger the further tree creation. Then it gives back the `outParent` stream to its parent. In total, all input streams are merged yielding `inAll`. Payloads aiming at the node itself are separated and handed over to the node function `nf` for further processing. The rest is merged with the output stream of the node function giving `inAll'` whose elements are routed to the fitting output stream. The first element sent to the outside, however, is the dynamic channel reference `href` for connecting the node to its partner. The routing itself is divided into two functions `routeN1` and `routeN2`. The first one is used while the hyper connection has not yet been established. It only uses the remaining ways for output. As soon as the `Ref` message of the partner node has been received, routing changes to `routeN2` which also uses the hyper node.

Two peculiarities about inserting hyper connections have also to be mentioned:

- After being defined, the input from the dynamic channel can be merged into `inAll` even if the connection has not yet been established. The hyper input stream is empty which does not harm the merging as merge is fair.
- While all hyper connections have not yet been constructed, all `Ref` messages for doing just that will be routed upwards to the father node and then downwards to their destinations. Hyper connections will be established approximately levelwise down from the father, as these `Ref` messages will reach their partners first. Subsequent messages can then be routed more and more via lower hyper connections. Therefore the the hypertree has an inherent starting phase as a binary tree where communication is expensive.

7.4.4 Applications

Now we will look first at a trivial example for illustration purposes and then at possible applications for the hypertree skeleton.

---

10 Example (Double Ping)
This trivial example just sends two messages to two grandchildren nodes and waits for their answers. These are counted via the father's state. The system terminates if both answers have arrived.

```
trivial :: Int -> [Int]
trivial h = outs
  where outs = hypertree h nf initstate updC updO term inps
```

```
hyperproc :: (Trans a, Trans b, Trans s) =>
  Int -> Code ->                              -- Current height, my code
  (Code -> [Msg a b s] -> [Msg a b s]) ->  -- Node function
  Process [Msg a b s] [Msg a b s]
hyperproc h me nf = process p where
  p inParent = side_eff `seq` outParent where
    -- Leaf ------------------------------------------------------------
    outPL = ...                                        -- analogous to node
    -- Node ------------------------------------------------------------
    (outPN, outLeft, outRight) = case route me pme ways1 of
      Up     -> (href:outPN', outL,      outR)
      Lefty  -> (outPN',      href:outL, outR)
      Righty -> (outPN',      outL,      href:outR)
    inLeft'  = cP (hyperproc (h-1) (leftChild  me) nf) outLeft
    inRight' = cP (hyperproc (h-1) (rightChild me) nf) outRight
    inLeft   = deLift inLeft'; inRight  = deLift inRight'
    [outPN', outL, outR, []] = routeN1 inAll' -- with internal hyper c.
    ------------------------------------------------------------------
    routeN1 :: (Trans a, Trans b, Trans s) =>            -- DISCONNECTED
               [Msg a b s] -> [[Msg a b s]]
    routeN1 []                    = [[],[],[],[]]
    routeN1 (State s       : ms) = [State s : ps, ls, rs, []]
      where [ps,ls,rs,[]] = routeN1 ms
    routeN1 (o@(Output b)  : ms) = [o : ps, ls, rs, []]
      where [ps,ls,rs,[]] = routeN1 ms
    routeN1 (p@(Payload src dst v) : ms) = tuple
      where tuple = fill p (routeN1 ms) (route me dst ways1)
    routeN1 (r@(Ref src dst c) : ms)
      | dst == me = parfill c hs2 [ps2,ls2,rs2,[]]
      | otherwise = tuple
      where [ps2,ls2,rs2,hs2] = routeN2 ms
            tuple             = fill r (routeN1 ms) (route me dst ways1)
    ------------------------------------------------------------------
    routeN2 :: [Msg a b s] -> [[Msg a b s]]                 -- CONNECTED
    routeN2 []                            = [[],[],[],[]]
    routeN2 (State s               : ms) = [State s:ps,ls,rs,hs]
      where [ps,ls,rs,hs] = routeN2 ms
    routeN2 (o@(Output b)          : ms) = [o:ps,ls,rs,hs]
      where [ps,ls,rs,hs] = routeN2 ms
    routeN2 (p@(Payload src dst v) : ms) = tuple
      where tuple = fill p (routeN2 ms) (route me dst ways2)
    routeN2 (r@(Ref src dst ch) : ms) | dst == me = error "Already c.!"
                                      | otherwise = tuple
      where tuple = fill r (routeN2 ms) (route me dst ways2)
```

Figure 7.18: Process definition within the hypertree

```
   -- Shared defs within where block --------------------------------
   side_eff       = (h==0)?((),inLeft' `seq` inRight' `seq` ())
   outParent      = (h==0)?(href:outPL,outPN)
   pme            = partner me
   (ch,inHyper)   = new (\ch val -> (ch,val))
   href           = Ref me pme ch
   inAll          = case h of
                      0 -> merge [inParent, inHyper]
                      _ -> merge [inParent, inHyper, inLeft, inRight]
   (inNF, inRest) = split (isPayloadFor me) inAll
   outNF          = nf me inNF
   inAll'         = merge [outNF, inRest]
   ways1          = (h==0)?([Up],      [Up,      Lefty,Righty])
   ways2          = (h==0)?([Up,Hyper],[Up,Hyper,Lefty,Righty])

fill :: Msg a b s -> [[Msg a b s]] -> Route -> [[Msg a b s]]
fill m [ps,ls,rs,hs] r | r == Up     = [m:ps,ls,rs,hs]
                       | r == Lefty  = [ps,m:ls,rs,hs]
                       | r == Righty = [ps,ls,m:rs,hs]
                       | r == Hyper  = [ps,ls,rs,m:hs]

(?) :: Bool -> (a,a) -> a
(?) True  (x,_) = x
(?) False (_,y) = y
```

Figure 7.19: Shared definitions within `hyperproc`

```
nf me ((Payload src dst val) : ps) =
  (Output (val+1)) : nf me ps
initstate = 0
updC s _  = s
updO s _  = s+1
term s    = s >= 2
inps :: [Msg Int Int Int]
inps = [ Payload father (leftChild (leftChild  father))  1,
         Payload father (rightChild (rightChild father)) 2 ]
```

Other applications include:

- **All-to-all communications**
  Every node and leaf is a process with equal rights. All-to-all communication is possible in an MPI-like way. This is the classical application of the hypertree: Whenever a set of processes has to communicate in an unexpected way, the Eden process tree is shortened by the hypertree.

- **Workpool**
  All leafs are workers, while all nodes are masters and submasters, respec-

tively. This gives $2^n - 1$ master processes and 2^n worker processes. As the tree is binary only, this is not yet as flexible as other workpool schemes. Hierarchical masters are immediately available.

- **Binary tree with level-wise rings**
  As shown by Goodman and Séquin[73] , this structure can easily be implemented using the hypertree skeleton.

The hypertree is universal enough to house different process schemes. It is a kind of generalised Eden process tree whose communication is made more efficient by wiring in bypasses of the most expensive communication pathways. The hypertree enables the programmer to quickly develop a process scheme at moderate costs; at the same time, the performance will also be moderate. Therefore the hypertree can be seen as efficient for some special schemes and as a rapid prototyping tool for many other schemes.

8. Developing Programs in Eden

"Programming is hard. [...] But parallel programming is much, much harder."

Simon Peyton Jones[80]

8.1 Motivation

The systematical development of functional programs is rarely handled in literature and research; see Chapter 4.1 of Thompson[202] for some hints. The cause for this may be that functional languages are considered being on such a high level of abstraction, that no specialised development method is necessary. This may be true for the development of small programs, but certainly not for the development of large ones. Introducing parallelism makes things even harder, as an additional dimension of complexity is introduced. In daily programming practice, programs are often the result of one of the two following development patterns:

- An application is given as a sequential legacy program, for which a parallelisation is wanted. After an inspection of the program's internals, it is essentially left unchanged but is punctually extended by side-effects for (often data-parallel) processes and demand control. The underlying algorithm is not really questioned for its suitability of being parallelised.
  - ⊕ If the algorithm is sufficiently parallelisable, a parallel version can be achieved quickly.
  - ⊖ However, if it is not, one has to undertake extensive rewriting of the program and to consider a complete reimplementation.
  - ⊖ Maintainability is limited because of the hack-like parallelisation.
  - ⊖ The resulting program can easily lack clarity and structure.
- The better alternative is to rewrite from scratch. But even then parallelism may be treated as a side-matter with sequential and parallel development steps being interleaved.
  - ⊕ Rapid development in the style of extreme programming[104] .
  - ⊖ No clear design decisions in the beginning.

⊖ Reduced maintainability and lack of clarity as before.

Our aim is to enhance the second way of program development by starting with a parallel core design and structuring the further development around that core. In our thesis we have presented until now only a loose collection of programming techniques for Eden which are more or less related to each other. In this chapter we will compose them to form a guide to developing programs in Eden. We will not construct a formal method of software engineering in the style of FAD[181] , but more a collection of good advice and best practice. This collection will be organised as a sequence of questions and advice guiding the successive development of an Eden program using the techniques presented. The chapter is divided into three steps and represents the sequence of development steps:

1. Choosing a parallelisation
2. Handling program phases
3. Building a program
   (a) Implementing in the large
   (b) Implementing in the small

In a top-down approach, we will describe a decision frame for constructing a typical Eden program. We will assume no special parallel architecture but that nodes are homogeneous (in terms of speed, memory, and network connection) and their number is known. The adaption to a more special architecture can later be done by hand, via special preprocessing steps, or by adaption (for example via automatic skeletons[79]).

> Section 8.2 starts program construction by choosing an adequate parallel algorithm which will dominate the program. Following that, Section 8.3 will describe how to handle and plan the five phases a parallel program goes through; these range from preprocessing to shutdown. Finally Section 8.4 enumerates techniques for parallel program construction. These are divided in two parts: first far-reaching programming decisions, and then programming details.

8.2 Choosing a Parallelisation

Given a problem that is to be solved, one first has to consult literature[108, 4, 175, 217, 146] to find a parallel algorithm for solving the problem. The first question concerning that algorithm is then:

1. Is the algorithm *transformational* or *reactive*? The first option means whether it calculates a result from an input in a mathematical sense; then termination can usually be initiated if the complete output has been produced. The second option describes algorithms which are not driven by computation but

by interaction. They set up process systems which react to events; termination is determined by state, which is either kept up to date in a central place or is collected from time to time. Reactive systems usually do not compute a value in the traditional sense.

A transformational system directly corresponds to the regular functional programming style; calculations are as usual with the exception of shifting selected evaluations to remote nodes via processes. However, if one aims at a reactive system, a good way to do so is to define:

- a process state and a state transition diagram
- a way of determining global state in a process as near to the root of the process tree as possible
- states which imply termination
- a communication diagram of interdependent streams together with the datatypes communicated via these streams

The workpool of Section 7.3 can be viewed as an example for such a reactive system, although it also possesses transformational behaviour. The next question is about the kind of control that drives the algorithm:

2. Is the algorithm *control-parallel* or *data-parallel*? Can a central control mechanism be identified or a central data structure from which parallelism emanates? Which is dominant?

Usually both aspects are true to some degree, as functional programs use control structures to transform values; both are therefore inseparable. Nevertheless, one usually dominates and a program orientation has to be chosen. As evaluation is determined by dependencies the next step is:

3. *Divide control or data* into the smallest possible independent or mostly independent parts.

This corresponds to the partitioning step of Foster's PCAM development scheme[64] for parallel programs. By doing so one breaks down control to the smallest possible bits.

4. *Regroup parts* to form *tasks* of an appropriate size. The right granularity has to be determined out of the number and speed of parallel nodes. As a rule of thumb, one should have more tasks than nodes, but not too many.

This step corresponds to Foster's agglomeration step. The next step is about the kind of tasks:

5. Are tasks of *regular or irregular size*? Is the task set *statically fixed or changing dynamically*? If tasks are developing dynamically, how is the task availability over time? Are tasks getting trivially available or are there strong dependencies releasing new tasks only one by one?

If a static set of regular tasks is given, the task distribution can be planned statically. If tasks are evolving dynamically or are irregular, a dynamic work distribution scheme like the workpool has to be chosen to gain load balance. We will now continue by defining processes:

6. Within the parallel algorithm, identify *central computations with little or better no interdependence*. These will later be turned into processes.

7. Is *explicit process placement* useful? Then define an explicit mapping via a function which maps process abstraction parameters to node numbers. Use this function when calling `createProcessAt`; it can easily be replaced by `-1` for stepping back to automatic mapping.

The next step is to decide the kind of communication, the structure of which is given by the dependencies between the processes:

8. Are values communicated as *single values* or are there *value streams* needed? If so, is non-determinism present? When merging streams, has *element order* to be preserved? If so, then values need to be paired with an unique identifier. Define datatypes which are communicated.

In the next Section we will deal with the phases the program will be in at runtime.

8.3 Handling Program Phases

In its lifetime from compilation to shutdown an Eden program passes through five phases, for each of which some things have to be noted:

- The *meta phase* is based on Template Haskell and can house the following actions: *Firstly*, the preprocessor shown in Chapter 3 will be run and will apply the eager transformation adding demand for let-defined process applications. This implicit demand bears the danger of interfering with the programmer's demand control structures; ideally, it should only add demand which already exists. Otherwise, one could choose to steer demand completely in an explicit way and switch off the transformation. By doing so one would avoid a mixture of implicit and explicit demand control. Furthermore, the programmer should be aware that other passes may be run on their code. The preprocessor will also add auxiliary functions and class instances to the program; name clashes may occur. *Secondly*, direct use of Template Haskell besides the preprocessor may happen which should not interfere. This also includes the use of automatic skeletons for adapting the program to the given parallel architecture.

- In the standard *compilation phase* Haskell transformations are applied to the Eden program. These include different optimisation steps[185] , which are: inlining, deforestation, and others. These may interfere with the intended Eden behaviour.

After the two compilation phases, three runtime phases follow. To illustrate them, we have taken the trace of Figure 7.10 and have inserted black lines to show them (see Figure 8.1). The phases are:

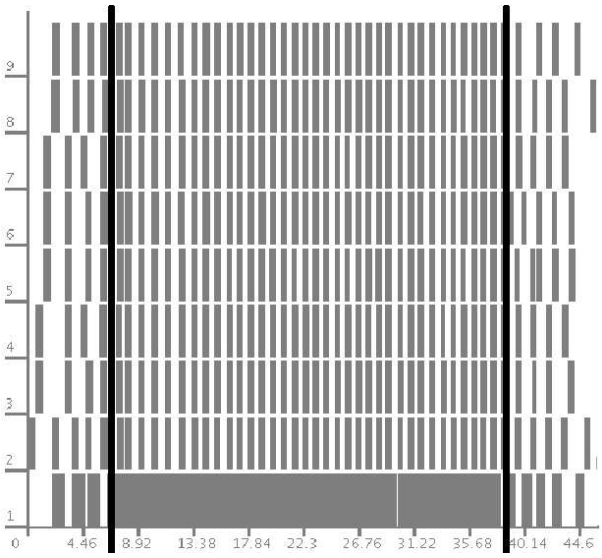


Figure 8.1: Runtime phases (from left to right): start, working, shutdown

- In the *starting phase* it is important to employ every parallel node as quickly as possible. This means that processes have to be created and that these processes have to be supplied with work. Therefore the direct mapping mechanism is useful, as the process abstractions carry their own tasks and will not have to wait for a second communication operation. It is also important to create processes in bulk and not one by one during evaluations. It can even be useful to create processes before their arguments are available; by doing so, the process creation has time to complete while its creator computes the process arguments. In the workpool of Section 7.3, the prefetch parameter is used to overcome the initial lack of tasks and to ensure a constant supply of tasks for each worker in the beginning.

  If tasks only evolve dynamically due to data dependencies, task shortage in the beginning cannot be avoided. This is the case in the workpool example of Figure 8.1, as the matrix contains data dependencies in the diagonal direction and the calculation starts with a single task. Task shortage is even worse, if (due to fast parallel nodes) granularity is further coarsened and tasks are united to form larger ones. If possible, one should try to keep the initial task set as large as possible while preserving a sufficient granularity. Under some circumstances, one may be able to statically compute some results to enlarge the initial task set.

- The *working phase* describes is the one where the system should run under (the highest possible) constant load. To achieve that, one should ensure the constant availability of communicated data in systems with interdependent streams. This can be done by using the techniques of Section 7.2; functions

preprocessing streams should be incremental to keep the streams flowing. Communication can be saved by sending container values which can carry multiple messages.

One should avoid the integration of process creations in functions which are called many times, as the constant build-up and shutdown of (practically identical) processes is too expensive. Instead, one should create a set of more general processes very early and supply these with different parameters to mimic the different versions.

Finally, termination has to be detected. In a purely transformational system, this most easily happens by running out of tasks; this will cause the processes to terminate. In a reactive system, this is usually more complicated as interdependent streams can block on each other waiting for more input. Therefore, as has been described in the previous Section, a state has to be kept. To avoid expensive algorithms for determining global state[146] , one can select a central process (usually high in the process tree) which monitors communication and keeps a representative state during computation. Often, such a local state is sufficient for detecting termination.

- The *shutdown phase* starts after termination has been detected. It consists of the result collection (and combination) for output and the shutdown of the process structure. The shutdown phase has to end quickly for the same reason as the startup phase.

  Usually the root of the process tree collects all results and detects termination. For output, results may need to be combined or transformed in the end. This is an inherently sequential task and should be avoided as far as possible. Transformations should be done by the processes in parallel. Additionally, the process structure should be one which supports quick result collection. It is also advisable to pipeline result collection by processing some results while others are being calculated.

  Like startup, shutdown contains inherent work which cannot be avoided. Processes have to be terminated, which happens either when there is no more input or no more need for output. In the latter case garbage collection will trigger termination.

In essence, one should create only a few processes, but these very early. Also, the root process should delegate as much work as possible to its child processes; not only in the starting phase (children calculate their arguments on their own), but also during shutdown (children help to transform their results).

8.4 Building a Program

Having decided the parallel algorithm, how to model it in Eden (tasks, type of parallelism, processes, data structures, termination), and how to handle the program phases, we can now go on with building the real program. We will construct an

Eden program in a top-down approach: First we will plan the general structure and then flesh out the details.

8.4.1 Implementing in the Large

In the large, we want to structure an Eden program into four parts which should only be intermixed if necessary:

1. *Datatype definitions and sequential access functions* can be defined first. They are the basic building blocks for function definitions and include

   (a) datatypes for sequential computations as well as

   (b) datatypes for communication and parallel coordination.

   Specific datatype implementations should be hidden behind access interfaces allowing for the implementation to change. For example, a general collection has a typical common set of access functions, but can be implemented (depending on its use) via lists, via a binary search tree, or other data structures.

   The preprocessor could be used to implement steps which derive a set of standard functions for each datatype in the same way it is done for our generic programming approach.

   If the datatypes are accessed not via the interface but directly, the later use of generic functions on processing these will tolerate changes in the data structures to some degree.

2. *Functions for sequential computation* comprise all functions needed for executing the basic computational part of the algorithm. They are based on the data structures and can ideally be tested in a sequential setting with no additional problems caused parallelism.

3. *Parallel coordination functions with demand control* should be defined separately as much as possible in the sense of aspect orientation. These deal with the creation of processes and channels and should not be intermingled with sequential code. These are mostly parallel skeletons together with some glue for connecting them, often *zip* and *unzip* functions for stream decomposition and demand steering calls. In general, side effects like process creation and demand control should be clearly separated in a function from the remaining code by executing them on block right at the beginning.

4. I/O is accessed only by the `main` function, `unsafe` I/O operations should not be executed. `main` is the gate for all input and all output and is responsible for exerting the correct amount of demand on the main parallel coordination function (often a skeleton) which triggers the remaining demand and control flow.

These parts also determine the order of development: At first the datatypes for the sequential computations and the parallel communication are defined together with their access functions. Based on these, the functions for the sequential computation are defined. At this point, the basic functionality of the program can already be tested, if the missing parallel parts are modelled via their sequential counterparts. Following that, parallel coordination and demand control are inserted. Now the orientation of parallelism is important:

Transformational systems come in general in two flavours which determine the construction of the parallel part:

- If *data-parallel*:

  For each datatype concerned, define a partitioning scheme corresponding to one defined in Section 8.2. Define a specialised one if a generic partitioning is too general. Insert (or replace existing sequential versions) generic data-parallel skeletons (either predefined or self-constructed ones) into the program. They can be placed either in separate functions with separate demand control and be called from within a sequential computation function. Alternatively, they can be inlined into the sequential code; in that case, however, a clear separation is useful and can be provided by triggering all effects on block before the main expression of a function.

- If *control-parallel*:

  Define datatypes for parallel coordination, which includes state messages, task and result messages, termination detection messages, termination messages, and others. Then predefined or self-constructed control-parallel skeletons are introduced in separate functions; these are called via demand control from a central function where the result of the skeleton is needed or from where it can be distributed.

Reactive systems are different in that a skeleton or a self-defined process structure incorporates a set of often interdependent streams which have to be connected to sequential computation functions. To set up the process structure, a communication diagram (like defined in the beginning and as shown in Figure 7.3) can mostly be translated into a structure of interdepending functions (see stream part of Figure 7.4). State can explicitly be communicated or implicitly by observing and interpreting tasks and results.

Finally, the `main` function is the source of all activity: After getting the arguments, these are given into a parallel coordination function which is called with a certain amount of demand, usually WHNF. From there, demand and parallelism propagates through the parallel functions which should result in almost all processes created in the beginning. These will then compute the results, which are given back to `main` for output.

Like in the spiral model[21] , the given sequence is not only traversed once but many times, as most probably the first parallel model will be in some way incomplete. The development of a parallel program is more an iterative process, during which a program usually proves to be too specialised. Datatypes may need to be extended or functions may need some generalisation to be able to cope with the

new datatypes. This can also happen because of granularity changes, which rise the need for functions being able to deal with a deeper datatype nesting.

When a satisfying program giving the correct results has been reached, it will contain generic parallel functions and will be in some sense architecture-independent. At this point, the program can be ported to various other parallel architectures or node constellations (with different nodes or varying network connections). Serving as a *program template*, it can for each architecture be specialised to a more specific program yielding higher performance: Generic functions can be transformed to their more efficient specialised counterparts, and meta-programming can be used to specialise the program to the given parallel architecture (in terms of task granularity, process distribution, and data distribution). For example, if the given parallel architecture is represented by a Haskell datatype like in Figure 3.18, one could partition data structures to fit this architecture (depending on the skeletons used). Additionally, skeleton parameters and process placements could also be chosen depending on that architecture description. This is similar to what Michaelson et al. do automatically in PMLS[78, 137] . All this is then part of the meta phase in Section 8.3.

8.4.2 Implementing in the Small

We will now turn to smaller implementation details. We will give a loose (an incomplete) list of advice groups; of course, all good programming habits of Haskell also apply to Eden. At first we will look at peculiarities of the given *architecture*:

- Due to the availability and price-performance ratio, clusters of standard PC nodes connected via ethernet have become by far the most common parallel architecture. A main characteristic of these is that they combine extremely fast nodes with modestly fast network connections. One therefore has to avoid communication as far as possible by combining small messages to larger ones.

Storage and laziness provide ample opportunity to loose performance:

- In Section 5.3 we have made clear that both in a sequential and in a parallel setting memory and performance can be lost if space leaks and large CAFs exist. Their existence will also trigger costly garbage collections. Also expensive is the typical stepwise programming, which applies a series of transformations not all at once but stepwise to a data structure; this causes a series of intermediate data structures to be allocated. This can be remedied by hand via function fusion or automatically via deforestation.

- When using data structures one is usually aware of its lifetime: Some are used throughout the whole program, others are only used locally and only a few times. The first are often called *persistent*, while the others are called *transient*. It is useful to keep in mind each data structures lifetime, as persistent ones are better fully evaluated while local ones should not.

Concerning *programming* in general a few things have to be noted:

- To obtain a general program, it is advisable to use higher-order and generic functions as much as possible to keep the program flexible to later changes and specialisations.

- Skeletons should have a common standardised interface to make it easier to replace their sequential counterpart. Figure 6.9 shows this for map: All parallel versions of map contain in their type at the end the type signature of map.

  With standardised interfaces, skeletons can also be plugged together to form a nested skeleton structure. This is also demonstrated by Kuchen[122] .

- When introducing data structures, one can begin with intuitive (and maybe inefficient) ones, separated by a clear interface (consisting of general access functions) from the remaining program. This means, that one could start modelling a set via a list which introduces bad access efficiency. When working with data structures, try to use generic functions to stay flexible. These can later be specialised to more efficient ones tailored to the final data structure implementation. In a parallel setting, generic partitionings are advisable for the same reason.

When writing a program, the programmer has many assertions and facts in mind which do not get written down in the program itself. While types are specified, other *knowledge is left implicit*. We propose to make that knowledge explicit:

- As has been argued in Section 5.6, we find it useful to explicitly note the demand behaviour of functions. In Eden we need to insert explicit demand control to trigger process creation, while at the same time it is unsure whether the triggering demand control expression is reached by enough demand. It is also useless to analyse the demand once during program construction, set up everything, and then leave behind a program which contains exactly the right amount of undocumented demand everywhere. Then any change or insertion would bear the risk of interrupting the correct demand flow.

  Together with the above mentioned standardised skeleton interfaces standardised explicit demand behaviours have to be introduced. Then it is clear whether a nesting of skeletons will trigger each other in a controlled way. To enable that, we will explicitly define that WHNF demand is needed to trigger side effects which precede the main expression of the skeleton, and that WHNF demand will be passed on to other skeletons. Thus we will gain a whole which only has to be touched by WHNF to unfold itself.

- Other meta information, like a data structure's lifetime and its degree of evaluation (see Figure 5.9), are important informations and can be kept on the value level where they can be used by the program. This can be a kind of explicit documentation on the program level.

9. Related Work

In this chapter we will list and shortly discuss related work. We will group the references according to the structure of our thesis.

9.1 Meta-Programming

Approaches to implementing meta-programming are numerous and we will only name the existing main directions and a few approaches:

Macro programming. Keith Wansbrough argues in "Macros and Preprocessing in Haskell" [215] , that meta-programming facilities are necessary also for Haskell. He mentions possibilities for program preprocessing in the style of the C preprocessor CPP as well as macro expansion.

Boost C++. By using templates, also C++[197] contains a kind of parametric polymorphism. As part of the Boost C++ libraries[1] , meta-programming via a preprocessor has also found its way into C++. Then even functional concepts like higher-order functions are expressible; the function `twice f = f . f` then looks like this:

```
template <class F, class X>
struct twice
{
  typedef typename F::template apply<X>::type once;     // f x
  typedef typename F::template apply<once>::type type;  // f (f x)
}
```

Generative programming. Component-based and generative programming aim at building software from standard components and are used in object-oriented imperative languages. They employ automatic source code creation through generic classes and templates; they are described in the standard reference by Czarnecki and Eisenecker [51] .

Pragmas. Related to this work is the approach by Peyton Jones et al. [167] , where so-called RULES pragmas can be inserted into the program (in the form of

special comments) which describe source-to-source transformations executed by the compiler. The compiler executes the transformations in the Core simplifier stage. For example, the *map*/*map* pragma fuses two maps into a single one:

```
{-# RULES
  "map/map" forall f g xs.
            map f (map g xs) = map (f . g) xs
#-}
```

The difference to our approach is that these are restricted to function applications and cannot work on expressions or whole programs. Still another alternative[203] is developed by Tolmach et al. which aims at defining a common *external Core syntax* format to help developers write external optimisation passes for the GHC. In total, pragmas are useful but limited in their power.

Partial evaluation. Program specialisation, also called partial evaluation, is also related to meta-programming. Such techniques enhance programs in terms of size or speed by precalculating results or parts of results. Classical works include papers by Futamura[68] and others[47, 114] .

Template Haskell. Template Haskell [189] (TH) is an approach to include basic meta-programming into the GHC. The work presented in Chapter 3 has been inspired by the work of Lynagh on Template Haskell. The first paper [144] shows five ways how Haskell could benefit from meta-programming: TH generates FFI interface code, TH generates class instances, TH removes boilerplate code, TH wraps optimisation code around functions, and finally TH is used for explicit program transformations. The second paper[145] elaborates on the fourth way by unrolling a recursive function to gain performance. Both papers discuss ways of manipulating Haskell code which are extended to a separate framework in this work.

Similar to our approach of outsourcing the preprocessor, Reid[177] has also used Template Haskell to turn the foreign-language interface Greencard[157] from a separate tool to a library. He also experienced considerably shorter code and an improved portability.

Torrano and Segura[204] also use Template Haskell to introduce passes for preprocessing programs. In particular, they show how to encode a reduced strictness analysis as well as a let-to-case transformation. They preassume, however, that the program to be preprocessed is already available as a value in their analysis program. Therefore, their approach lacks a connection between the preprocessing passes and the host compiler.

Seefried et al. investigate an approach[186] similar to ours: Template Haskell is used to implement and optimise an extension of a host language.

Scheme. Scheme has a long tradition of macro programming. *Hygienic macros*[43] have been introduced to avoid easier expansion avoiding name-capture. For a discussion see Sheard and Peyton Jones[189] .

Multi-stage Programming. While we have introduced static meta-programming into Eden, other approaches even allow for dynamic meta-programming which is called *multi-stage programming*[199] ; by doing so, partial evaluation and program specialisation techniques are provided directly to the programmer. MetaOCaml[200] , which is based on the strict ML[163] , is an example of a multi-staged language. It provides basic annotations (brackets, escape, and run) for delaying and forcing the execution of an expression.

Research on this approach is also done by Christoph Herrmann and Christian Lengauer at the University of Passau in Germany. They also use Template Haskell and MetaOCaml and even apply meta-programming to the construction of parallel programs[89] . There, parallelism is introduced into general skeletons while the programmer does not need to have knowledge of parallelism. Meta constructs are used for the generalisation of the skeletons.

The general idea of 'separation of concerns' has been carried on by Veldhuizen et al. in their work on *active libraries*[50, 209] . This relates to our approach of separating host compiler and extension implementations by pulling the latter into a separate meta-programmed library.

9.2 Generic Programming

The term *generic* can be misleading as it has been used in many different contexts. It is also known as *polytypic* or, what seems to be the most clear term, *structural polymorphism*. For the extensive research done in that area see in general the excellent and comprehensive bibliographies of Hinze [95] and (the more recent) Löh [136] . For a general motivational article see Backhouse et al. [12] .

Derivable Type Classes/DrIFT. Building on Hinze's earlier work[102] , Hinze and Peyton Jones introduced *derivable type classes*[103] into Haskell. In this article, type classes are enriched with the possibility to define very general default methods over a sum-of-products representation (in essence structural induction) of the given datatype. For an empty instance declaration of such a class, these general default methods are then used to generate the corresponding datatype-specific methods. This means, that a generic function is defined via general default methods; like a template, it can then be used to construct instances for various types. The approach is, however, restricted to datatypes of kind `(*)`. In a sense, this mechanism can replace Haskell's `deriving` mechanism (which in itself can be seen as a very simple form of genericity). For the sake of completeness, Alimarine and Plasmeijer have given a similar mechanism[5] for Clean.

The automatic generation of instances can also be done via the DrIFT[216] preprocessor, which is however a separate tool. In DriFT, instance generation is specified via Haskell functions which map type representations to a Haskell code representation.

PolyP. Following the work[150] of Meertens on polytypy, Jeuring and Jansson have developed an approach to generic programming called *PolyP*[111, 112] . Representing regular (no higher-kinded or nested) datatypes via polynomial pattern functors and initial algebras, they define polytypic functions over a sum-of-products representation of the given argument datatype. As this corresponds to structural induction over the datatype, polytypic functions are most often defined via a catamorphism and a polytypic map operator. For being able to work on the general representation of a datatype, two functions `in` and `out` for making the representation transition are needed.

The polytypic functions are collected into the PolyLib library, which is described in detail both in Jansson's PhD thesis[109] as well as in the extensive introduction[12] by Backhouse et al. A more recent port[158] of PolyP to Haskell 98 is based on the same principles: It first transforms a value into an general representation, applies the generic function to that representation, and then transforms the result back into the Haskell world. The transformation functions are contained in instances of the `FunctorOf` class, which have to be written by hand or derived automatically via some unspecified tool.

Norell and Jansson also compare and implement[159] the two quite similar approaches PolyP and Generic Haskell (see next item) in Template Haskell.

Generic Haskell. Generic Haskell evolved out of a long series of research papers. It started with Hinze's work[102, 94] on a generic programming extension for Haskell, which we have already described in the context of derivable type classes above. In essence, type classes are interpreted as collections of potentially generic function type signatures which are specified via inductively defined default methods. Instances are then the actual specific implementations with respect to the instance datatype. These generic functions can also be called type-indexed, as they are defined over a range of types. However, the kind of these types must be flat, which includes `(*)`, `(* -> *)`, and so on. In contrast to the PolyP approach, pattern functors do not appear; generic functions can be directly defined over the inductive datatype structure. The theoretical framework is developed in another paper[93] . This approach, like ours, is not flexible in terms of kind.

This is remedied in a later extension[96] , in which it is possible to define a single generic function for all kinds (meaning it is also kind-indexed). In his habilitation thesis[95] , Hinze compares both approaches and selects the second one for the basis of *Generic Haskell*. He also describes a way of implementing the approach: For every datatype encountered in a program that is applied to the generic function, a specific instance of that function is generated. This is similar to our approach, although we generate an instance for every datatype contained in the program without doing a usage analysis. General and extensive overviews about practice, theory, and applications[98, 97] are also available.

Generic Haskell has also been implemented, and practical experience has led to improvements[41, 99] which were implemented in a following release. Dependency-style Generic Haskell[135] enhances the syntax and provides more

flexibility in Generic Haskell. The theoretic foundations until here as well as implementation aspects are covered in the PhD thesis[136] of Löh.

Stratego/Strafunski. In term-rewriting it is common to implement and use tree-traversals via combinators. *Stratego*[210, 211] is such a language, which provides primitives for generic tree-traversal; conditional rewrite-rules are then used to modify the traversed syntax tree. This is also closely related to our approach of implementing our preprocessor in Chapter 3.

Tree-traversals can also be used to implement generic traversals into subterms of a structure. In the so-called *Strafunski* approach, Lämmel and Visser present "Typed Combinators for Generic Traversal"[128] . These are organised as strategies[126, 127] , which besides allowing for generic traversal also provide non-generic ad-hoc behaviour for specifically selected types. This approach is ported to Haskell in the series of boilerplate papers.

Boilerplate. Code for the traversal of data structures, also called boilerplate code, is considered tiresome and errorprone in writing. Using the Strafunski approach, Lämmel and Peyton Jones[123] show a way to "scrap your boilerplate" in Haskell (SYB for short). They show how to generically traverse arbitrary data structures while having specialised functions being applied if a specified subterm is encountered. The approach is presented as a combinator library. The necessary class instances are created by the DrIFT tool mentioned above. The whole approach is, however, quite element-oriented and does not allow access to the structure.

A followup paper[124] provided generic buildup in contrast to the generic traversal and consumption. Another one[125] changes the cast-based implementation to one based on type classes.

Hinze, Löh, and Oliveira have reimplemented[101] SYB in Haskell. They explicitly define a `Type` datatype as a generalised algebraic datatype (GADT for short) which represents types which can be built out of the given elements of `Type`. Using explicit typing of values, overloaded functions can be implemented[70] without type classes; this is not yet truly generic, but very flexible. Then a `Spine` datatype is used to lift explicitly typed values to an easily accessible spine level and back using the functions `toSpine` and `fromSpine`. Then truly generic functions can be written working on that spine representation. Nevertheless, for a new datatype it is still needed to extend the `Type` datatype as well as the `toSpine` function.

This approach is similar to ours, as both access the spine of a data structure to be able to let generic functions work on it. The given approach deals with overloading by introducing explicit types and GADTs, while we use type classes. Type classes introduce additional type contexts which may be tiresome, but one has not to give the correct type of an argument when using a generic function. It is however a natural and elegant solution to make types directly available to the program by coding them explicitly as values. Another advantage is that arbitrary kinds can be handled.

Like SYB, the spine view is element-oriented, which makes it easy to consume structures but hard to construct structures. Our approach shares this deficiency; but as we aim at parallel programming, deconstruction is much more important than construction. For SYB, this limitation is remedied by Hinze and Löh[100] . The type spine view makes it possible to define generic producers. It remains future work to extend our approach accordingly.

Charity, FISh, G'Caml. Following Norell and Jansson[158] , other non-Haskell approaches include Charity[44] , FISh[110] , and G'Caml[67] . We will not further describe these here.

9.3 Demand Control

Demand control has already been recognised as an important topic by Hughes in his PhD thesis [106] . He was the first to describe space leaks caused by lazy evaluation and described the `seq` function as one possibility to overcome (some kinds of) them.

Harrison and Kieburtz describe in their paper "Pattern-Driven Reduction in Haskell" [85] , how pattern-matching (contained in function definitions, `let` expressions, and `case` expressions) affects non-strict evaluation. They also describe the semantics of evaluations enforced by pattern-matching. In a sequel paper "The Logic of Demand in Haskell" [86] , they introduce a programming logic which formalises the mixed evaluation (partly lazy, partly strict) of pattern-matching to ease specification and verification of Haskell programs.

Harrison, Sheard, and Hook set out to examine the difference between the λ-calculus and the mixed evaluation order of Haskell in their paper "Fine Control of Demand in Haskell" [84] . There they introduce a calculational semantics for Haskell which exposes the interaction of the default laziness with Haskell's strict features.

In his paper "Eager Haskell: resource-bounded execution yields efficient iteration" [147] , Maessen proposes to evaluate Haskell eagerly by default to circumvent space leaks and introductions of `seq`. Only if resource bounds (stack, heap, or time) are exceeded, computation is switched back to lazy evaluation.

Evaluation strategies, as already presented in Figure 5.5, expand the `seq` operator inductively over data structures for different depths of evaluation. Strategies have been introduced by Trinder et al.[205] and the language containing them has extensively be compared to Eden in "Comparing Parallel Functional Languages: Programming and Performance" [137] .

9.4 Data and Control Parallel Skeletons

The term *skeleton* has been coined by Cole[45] . The general understanding is, that a skeleton is an abstract language construct (like a higher-order function or a tem-

plate) which encapsulates a useful pattern of implicit or explicit parallel cooperation.

Skeletons have also been integrated into imperative languages; we name a few approaches:

Skil. Skil[22] is an imperative language, enhanced with functional features (higher-order functions, currying, polymorphic type system) and parallel skeletons. The concepts of Skil have been extended to C++ by Kuchen[122] .

Poldner and Kuchen[170] argue, that in a normal workpool the master process has to both distribute work and collect results. Therefore they propose the division of the master into two processes, a *dispatcher* and a *collector*. One is responsible for distributing work while the second one only has to collect results. Therefore each of them is burdened less than before. A clear disadvantage is however, that the distributor is spreading work randomly as he has no knowledge about the worker's loads. Poldner and Kuchen argue that by random assignment the load is roughly kept equal.

SAT. The SAT approach[74] by Gorlatch and Lengauer stands for staging and transformation, as the approach is about gaining internally parallel stages by formal program transformations. It focuses on a single high-level parallel programming pattern, the list homomorphism, to capture the divide-and-conquer skeleton. Ported to Haskell, this approach was called HDC[88] , which was implemented in C and MPI.

Skeletons have also been modified for being used in a high-latency Grid environment[6] . There, Java is enhanced by the introduction of parallel skeletons.

P3L. P3L [53, 52] integrates parallel skeletons (farm, pipe, map, reduce, loop) into the host language C++. Furthermore it allows for the nesting of these skeletons. Every skeleton is accompanied by an implementation template which can be specialised to a given parallel architecture.

With higher-order functions, functional languages have for some time been used as a target for the introduction of parallel skeletons; fundamental work has been done by Darlington et al.[54, 55, 56] . General overviews (Chapter 13 of Hammond and Michaelson[80] , the book of Rabhi and Gorlatch[176] , and a Dagstuhl seminar on high-level parallel programming [46]) show the wide range of approaches. We name a few of them:

SkelML. In his PhD thesis[23] , Tore Bratvold has integrated skeletons in the strict functional programming language ML. Programs are analysed and parallelism is automatically extracted, most often directly out of the use of higher-order functions (like map, filter, and fold). Performance models for each skeleton help decide if a parallelisation is promising an efficiency gain or not.

PMLS. The PMLS system[78] is a direct successor of the SkelML system and has also been developed at the Heriot-Watt University in Edinburgh.

HaskSkel. In HaskSkel[81] , Hammond and Rebón Portillo combine evaluation strategies of GpH[205] with the efficient data structure implementations of Okasaki's Edison library[162] .

Eden. We have shown Eden skeletons in our thesis, but a paper[69] containing a collection of basic skeletons (like pipeline, divide-and-conquer, grid, and others) has also to be mentioned.

9.5 Developing Programs in Eden

It seems that not much research has been done in the area of systematic program development of (parallel) functional programs. There are lots of technical tools like integrated development environments [8] , version control systems[178] , debuggers[40] , testing tools[42] , automatic documentation tools[148] , interfaces to foreign languages[177] , graphical user interface libraries[192] , and profilers[184, 117] . These are undoubtedly useful, but we aim more at approaches for developing programs in the sense of software engineering. The traditional approaches all mainly aim at developing much larger systems:

Waterfall. The classical *waterfall model* of Royce[179] defines program development to flow down the steps of requirements specification, design, implementation, verification, and maintenance. However, it assumes that every phase is fully completed, before the next one is begun; it is considered *non-iterative*. In practice, this is hard to achieve and changes are hard to integrate. The idea of rapid-prototyping is also not included.

Spiral/RUP. In contrast, the *iterative* (sometimes also called incremental) development method connects the two ends of the waterfall to a cycle. This means, that evaluations during maintenance give rise to new requirements and lead to a new iteration in program enhancement. The waterfall is applied iteratively and yields an evolutionary design process. This has also been done by Boehm in his *spiral model*[21] . A successor of this model is *Rational Unified Process (RUP)*[121] . These iterative methods allow for rapid-prototyping.

Agility/XP. In these approaches[104] , it is accepted that requirements and deliverables are subject of *constant* change throughout the whole development process. Therefore, being able to adapt to changes is considered more important than achieving a full specification in the beginning.

Beneath these, we have only been able to identify the following more specialised approaches for functional languages:

FAD. The most important and most complete approach to functional program development is *FAD*[181] . FAD stands for *Functional Analysis and Design* representing its main two phases which consist of a requirements analysis and an implementation design phase. In the first phase, requirements and deliverables are analysed and defined. The second phase then aids the construction of a functional program. All these processes are expressed in a pictorial

language, which represents each component of a functional program as a diagram which can be linked together. This is a very complex, complete, and model-oriented approach and hardly comparable to our more practical design guide.

Program Derivation. As a part of the development of complete programs, specialisation via function transformation is well-known. Duality and fusion theorems in the style of Bird[17] allow for transforming a naive function implementation into one much more efficient.

Refactoring. Refactoring (see the *Haskell Refactorer HaRe*[133] for example) can be seen as one part of program development: It deals with redesigning and enhancing programs without changing their semantics. Refactoring steps often correspond to modifications every programmer does by hand from time to time; for example, consider the process of generalising a function by introducing an additional parameter (introducing a step parameter to the `from` function):

```
g x = x : (g (x + 1))      ->      g n x = x : ((g n) (x + n))
```

Existing calls to `g` have also to be replaced:

```
g z                        ->      g 1 z
```

Steps like these occur often during program construction, and can (if done manually) introduce harmless but annoying bugs. Automatic rewriting therefore improves the situation.

Refactoring steps can be implemented as static preprocessing steps. Then also the Eden eager transformation step is part of a refactoring process. Although being useful, refactoring is not a complete development method but only an enhancement method for existing programs.

Of course, all these apply also to parallel functional languages. Specialised approaches, however, are:

GpH. For Glasgow parallel Haskell, a development scheme together with some applications has been given in [138] . It is based on having a correct sequential implementation and then incrementally parallelises this implementation aiming at achieving moderate parallelism at low cost. Firstly, a thorough analysis and a time profile has to reveal program hotspots and top-level parallelism. These hotspots are then parallelised using evaluation strategies[205] which are essentially skeletons combined with demand control. The result is then tested via the simulator GranSim[139] and refined if necessary. Then tests on real systems are run to prove the efficiency of the parallelisation.

10. Conclusion

10.1 Conclusion

To structure our conclusions, we recall the structure of our thesis from the beginning in Figure 10.1. Our thesis described techniques for meta-programming, generic programming, and demand control. Building on these, skeletons and functions for data-oriented and control-oriented parallelism have been given. Shaped like a funnel, all this is then concentrated in an outline of how to develop programs in Eden.

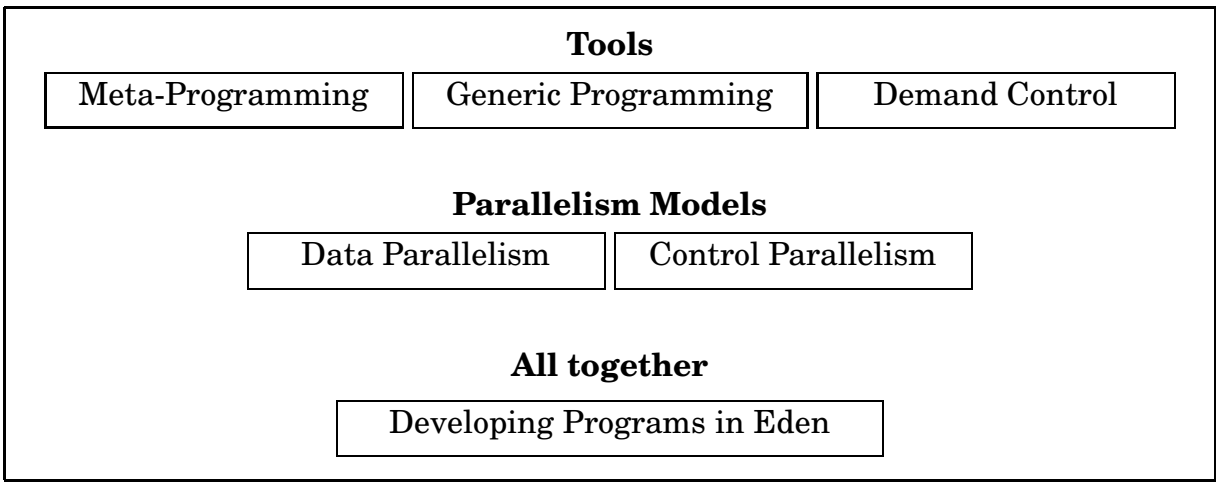


Figure 10.1: Structural Overview

The preprocessing tool presented allows for the definition of Eden's domain-specific preprocessing passes as an external active library. This is typical in domain-specific languages, which often extend a standard base language by a special topic: Being implemented as an extension of an also often standardised base language compiler, the portability of these implementations benefits greatly from leaving the standard compiler as untouched as possible. This is what we have partly done for Eden, which is a parallel extension of Haskell based on the Glasgow Haskell Compiler. The preprocessor has been implemented via Template Haskell, which unfortunately still is subject to change. Of course the whole construction relies on the future availability of Template Haskell in GHC.

Inside the tool we have defined a stateful monadic traversal of the abstract syntax tree. This traversal allows for the flexible redefinition of syntactical constructs,

as well as global redefinitions. We have used it for deriving class instances as well as for program transformations. The quasi-quotation operator has not been constructed for our use, but works nevertheless besides misinterpreting layout at times. This can be remedied by using explicit parentheses at the moment.

An approach to generic programming aimed at supporting parallelism needs to be structure-oriented, which means that data structures have to be accessible layerwise. This is necessary, as parallelism relies on traversing these for creating processes connected to the correct layers as well as for partitioning and recombining argument data structures. We have given such an approach based on type classes, which allows to access data structure layers via constant de- and reconstruction. We have called our approach reduced, as it is not as general as other approaches; as a main downside, we cannot handle arbitrary kinds inductively. On the other hand, this is not very often necessary, since more than two type variables are rarely met. Another downside is the overhead caused by de- and reconstruction. This is alleviated by the fact, that mostly quite small data structures are handled generically in a parallel setting. This is even further alleviated, as we consider our approach as a means for rapid parallel prototyping, aiming more at the quick construction of a correct program than at competing with a most cleverly constructed hand-crafted one. In our approach a more efficient program is constructed in a sequence of successive enhancements, including the specialisation of generic function application to specialised and more efficient ones.

The control of demand in a lazy parallel language is vital for gaining speedups. We have summarised methods for demand control and categorised them as data-oriented and control-oriented. For the first topic we have given generic versions of the important spine evaluation with branch selection. For the second topic we have identified and separated effects in a function. We have shown how to express these uniformly and how to group them.

The difference between data- and control-orientation carries over to parallel control. Data-parallel methods rely on the partitioning and recombining of argument data structures. We have given generic methods to do that. Then we have turned to parallelising the *map* function. We have generalised three data-oriented parallelisations, and have looked at combining these for nested data structures. Control-parallel methods do not naturally leave much room for genericity. We have identified three topics for which we have provided solutions: Firstly, we have shown techniques to deal with the omnipresent streams in Eden. Secondly, we have applied these techniques to deal with irregularly sized tasks in the workpool skeleton. The workpool skeleton has been presented in three flavours, the basic workpool, one capable of dynamically managing new tasks, one for introducing a hierarchy of master processes, and one with a stateful master. Lastly, we have constructed a Hypertree skeleton for cutting short long communication distances in the Eden process tree.

As a result of all the techniques shown so far, we have set out to summarise all this to form a guideline for Eden program construction. This guideline is an instruction manual on Eden programming, and not a formal program derivation calculus like the transformational one by Bird[17] . Nevertheless we think that this could be a good starting point for developing a more sophisticated program development methodology. In total, we have introduced new techniques into parallel

functional programming, which will hopefully lead not only to faster but also to better programs.

10.2 Future Work

This thesis covers a wide range of programming techniques, which all leave room for further investigations. We will outline a few ideas for future work in the order described in Figure 10.1:

- The preprocessing tool shown in Chapter 3 has to be integrated as a standard part of the Eden implementation. Together with the work of Jost Berthold, who reimplements the Eden runtime system in Haskell, significant parts of the Eden implementation are then expressed in Haskell. The result will be a much better versional portability.

  Furthermore, reflection and adaption are two extremely useful techniques. For both, new preprocessing steps have to be devised. One can think of many reflections, like:

  - analytical steps which execute a non-determinism analysis[187] or an analysis about the usage of dynamic channels (single use only)
  - informational steps which give an abstract view of the program, marking central functions and hotspots where parallelism and memory usage concentrate (see Medina[182] for example)
  - transformational steps which, like the eager transformation, optimise predefined expression constellations to support the common programming style (which could result in inefficiency) with parallel efficiency

  The same is true for adaption: self-configuring skeletons have to be further investigated, as the adaption to the many and very different levels of hardware parallelism (with the associated memory hierarchy) raises the need for very adaptive skeletons.

- Generic programming also leaves much room for improvement. The approach described in this thesis is based on type classes. Recent improvements of the type system open new possibilities of implementing generic techniques; therefore, the principle of de- and reconstruction could be reimplemented in a more flexible way. Especially the recent research[100] of Hinze and Löh on type spine views is interesting. Also, a library of sequential and parallel generic functions has to be constructed; in this thesis, we have given only a couple of generic functions. Finally, much more programming practice with generic functions in a parallel context is needed, especially for larger applications. Then we can see, if the overhead introduced by our approach is outweighed by advantages in program development. Also the specialisation of generic skeletons to a data structure can be implemented as a preprocessing step. The insights gained that way are needed for further improvement.

- Regarding data parallelism, more research can be done on finding appropriate generic partitioning methods. Also, the usefulness of generic partitioning has to be assessed; in a larger program context, they may prove to be too general. To accompany *map*, other functions of the BMF[18, 193] and Squiggol community (like *fold* and *filter*) could also be a target for parallelisation.

- Naturally, genericity tends more towards data parallelism than towards control parallelism. Nevertheless, more research has to be done on how generic functions can be useful in control parallel skeletons. Furthermore, more standardised solutions (in the form of skeletons) have to be found for other typical constellations in parallel programming. In a form of a library with exactly defined demand, storage, and parallel behaviours, these will support our plug-and-play approach to program development.

- Systematic program development methods for parallel functional languages, even for only functional ones, hardly exist. This is a widely open field of research, and we have merely sketched one approach of doing so by summarising our experiences and techniques in a unified context.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2005.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT Press, 1986.

[3] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1993.

[5] A. Alimarine and R. Plasmeijer. A generic programmin extension for Clean. In *Proceedings of the 13th International Workshop on the Implementation of Functional Languages (IFL'01)*, volume 2312 of *LNCS*, pages 168–185. Springer-Verlag, 2001.

[6] M. Alt, H. Bischof, and S. Gorlatch. Program development for computational grids using skeletons and performance prediction. *Parallel Processing Letters*, 12(2):157–174, 2002.

[7] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[8] Krasimir Angelov and Simon Marlow. Visual Haskell: A full-featured Haskell development environment. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, Tallinn, Estonia*, pages 5–16, September 2005.

[9] J. Anvik et al. Generating Parallel Programs from the Wavefront Design Pattern. In *Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'02)*, April 2002.

[10] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[11] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice-Hall, 2nd edition, 1996.

[12] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming – An Introduction –. In S. Doaitse Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Advanced Functional Programming, LNCS 1608*, pages 28–115. Springer-Verlag, 1999.

[13] H. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[14] J. Berthold. Towards a generalised runtime environment for parallel haskells. In Marian Bubak et al., editors, *Computational Science — ICCS'04*, volume 3038 of *LNCS*, page 297ff, Krakow, Poland, 2004. Springer-Verlag.

[15] J. Berthold, U. Klusik, R. Loogen, S. Priebe, and N. Weskamp. High-level Process Control in Eden. In Harald Kosch et al., editors, *EuroPar 2003 – Parallel Processing*, volume 2790 of *LNCS*, Klagenfurt, Austria, 2003.

[16] J. Berthold and R. Loogen. The Impact of Dynamic Channels on Functional Topology Skeletons. In *Proceedings of the 3rd International Workshop on High-Level Parallel Programming and Applications (HLPP)*, 2005.

[17] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2nd edition, 1998.

[18] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.

[19] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (3.1). Technical Report CMU-CS-95-170, Carnegie Mellon School of Computer Science, USA, September 1995.

[20] OpenMP Architecture Review Board. OpenMP Application Program Interface, 2005. (see also `http://www.openmp.org`).

[21] Barry Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[22] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996.

[23] Tore A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, November 1994.

[24] Thomas Bräunl. *Parallele Programmierung: Eine Einführung*. Vieweg, 1993. In german.

[25] S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Towards a declarative language for concurrent and parallel programming. In *Glasgow Workshop on Functional Programming 1995*. Springer-Verlag, 1995.

[26] Silvia Breitinger. *Design and Implementation of the Parallel Functional Language Eden*. PhD thesis, Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, June 1998.

[27] 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-Verlag, 1997.

[28] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Functional Programming 1997*, 1998. revised version.

[29] 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)*, Springer LNCS, 1998.

[30] 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 Functional Languages 1997, St. Andrews, selected papers*, LNCS 1467. Springer-Verlag, 1998.

[31] Silvia Breitinger and Rita Loogen. Explicit Process Systems in Eden. In *Constructive Methods for Parallel Programming (CMPP), Marstrand, Sweden*, 1998.

[32] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.

[33] Silvia Breitinger, Rita Loogen, and Steffen Priebe. Parallel Programming with Haskell and MPI. In *Proceedings of the Conference on the Implementation of Functional Languages, London 1998*, 1998.

[34] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, January 1965.

[35] Geoffrey Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. PhD thesis, Imperial College, London, 1987.

[36] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60. IEEE Computer Society, April 2004.

[37] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[38] Brad Chamberlain. An Introduction to Chapel: Cray's High-Productivity Language, September 2005. Talk at the Parallel Global Address Space (PGAS) Programming Models Conference, Minneapolis, USA.

[39] J. Cheney and R. Hinze. A Lightweight Implementation of Generics and Dynamics. In M. Chakravarty, editor, *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*, October 2002.

[40] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL 2000), Aachen, Germany*, pages 176–193. LNCS 2011, Springer-Verlag, September 2001.

[41] D. Clarke and A. Löh. Generic Haskell, specifically. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, number 115 in International Federation for Information Processing, pages 21–47. Kluwer Academic Publishers, 2003.

[42] K. Classen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming (ICFP 2000), Montréal, Canada*, September 2000.

[43] W. Clinger and J. Rees. Macros that work. In *Proceedings of the 19th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'00)*, pages 155–162. ACM Press, January 1991.

[44] R. Cockett and T. Fukushima. About Charity. Technical Report Yellow Series Report No. 92/480/18, Department of Computer Science, University of Calgary, Canada, 1992.

[45] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman / MIT-Press, 1989.

[46] M. Cole, S. Gorlatch, J. Prins, and D. Skillicorn. High level parallel programming: Applicability, analysis and performance. Technical report, Dagstuhl-Seminar-Report 99171, April 1999.

[47] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[48] D. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann Publishers, November 1998.

[49] José C. Cunha and Omer F. Rana. *Grid Computing: Software Environments and Tools*. Springer, December 2005.

[50] K. Czarnecki et al. Generative Programming and Active Libraries. In *Dagstuhl Seminar on Generic Programming*, volume 1766 of *LNCS*, April 1998.

[51] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, first edition, June 2000.

[52] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in P3L. In *Euro-Par 1997, LNCS 1300*, pages 619–628. Springer-Verlag, 1997.

[53] Marco Danelutto and Susanna Pelagatti. Parallel Implemenation of FP using a Template-based Approach. Technical report, University of Pisa and University of Edinburgh, 1995.

[54] 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. Technical report, Imperial College, London, 1992.

[55] John Darlington, Yike Guo, Hing Wing To, and Jin Yang. Parallel Skeletons for Structured Composition. Technical report, Imperial College, London, November 1993.

[56] John Darlington, Yike Guo, Hing Wing To, and Jin Yang. Functional Skeletons for Parallel Coordination. In *Proceedings of Euro-Par 1995*. LNCS 966, Springer-Verlag, 1995.

[57] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[58] J. G. Delgado-Frias, J. Nyathi, and D. H. Summerville. A programmable dynamic interconnection network router with hidden refresh. *IEEE Transactions Circuits and Systems*, 45(11):1182–1190, November 1998.

[59] Kemal Ebcioğlu, Vijay Saraswat, and Vivek Sarkar. X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access. In *Proceedings of the 3rd International Workshop on Language Runtimes (Part of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA)*, 2004.

[60] H. Ehrig, B. Mahr, and F. Cornelius. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag, 2001.

[61] J. T. Feo. A Comparative Study of Parallel Programming Languages: The Salishan Problems. In *Special Topics in Supercomputing*, volume 6. North Holland, 1992.

[62] M. J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[63] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[64] Ian Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

[65] Ian Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1989.

[66] G. Fox. Parallel computing comes of age: Supercomputer level parallel computations at caltech. *Concurrency - Practice and Experience*, 1(1):63–103, 1989.

[67] J. Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, 2001.

[68] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler systems. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[69] 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.

[70] J. Gibbons and O. de Moor, editors. *The Fun of Programming*, book chapter Fun with Phantom Types by Ralf Hinze, pages 245–262. Palgrave, 2003.

[71] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Third International Conference on Functional Programming (ICFP)*, pages 273–279, 1998.

[72] P. Roldán Gómez. Eden Trace Viewer: A Tool to Visualize Parallel Program Executions. Diploma thesis, Universidad Complutense de Madrid, 2004. In german.

[73] J. R. Goodman and C. H. Séquin. Hypertree: A multiprocessor interconnection topology. *IEEE Transactions on Computers*, 30(12):923–933, 1981.

[74] Sergej Gorlatch and Christian Lengauer. Abstraction and Performance in the Design of Parallel Programs: An Overview of the SAT Approach. *Acta Informatica*, 36(9/10):761–803, 2000.

[75] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*. Addison-Wesley, 3rd edition, June 2005.

[76] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.

[77] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type Classes in Haskell. In *European Symposium On Programming, LNCS 788*. Springer-Verlag, April 1994.

[78] M. Hamdan. *A Combinational Framework for Parallel Programming Using Skeletons*. PhD thesis, Department of Computing and Electrical Engeneering, Heriot-Watt University, Edinburgh, January 2000.

[79] K. Hammond, J. Berthold, and R. Loogen. Automatic skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, September 2003.

[80] K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*. Springer-Verlag, November 1999.

[81] K. Hammond and Á. J. Rebón Portillo. HaskSkel: Algorithmic Skeletons in Haskell. In P. Koopman and C. Clack, editors, *Proceedings of 1999 Workshop on the Implementation of Functional Programming Languages*, volume 1868, pages 181–198. Springer-Verlag, 2000.

[82] Kevin Hammond. Parallel Functional Programming: An Introduction. In *PASCO, Linz, Austria*. World Scientific, 1994.

[83] Kevin Hammond, James Mattson, Jr., Andrew Partridge, and Simon L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In Thomas Johnsson, editor, *Workshop on the Implementation of Functional Languages, Båstad, Sweden*. Chalmers University of Technology and University of Göteborg, September 1995.

[84] W. Harrison, T. Sheard, and J. Hook. Fine Control of Demand in Haskell. In *6th International Conference on the Mathematics of Program Construction*. Dagstuhl, Germany, 2002.

[85] W. L. Harrison and R. B. Kieburtz. Pattern-driven Reduction in Haskell. In *2nd International Workshop on Reduction Strategies in Rewriting and Programming (WRS02), Copenhagen, Denmark*, 2002.

[86] W. L. Harrison and R. B. Kieburtz. The Logic of Demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, November 2005.

[87] John L. Hennessy and David A. Patterson. *Rechnerarchitektur*. Vieweg, 1994. In german.

[88] Christoph Herrmann and Christian Lengauer. HDC: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters*, 10(2/3):239–250, 2000.

[89] Christoph Herrmann and Christian Lengauer. Functional Metaprogramming in the Construction of Parallel Programs. In *Proceedings of the 4th International Workshop on Constructive Methods for Parallel Programming (CMPP'04), Stirling, Scotland, UK*, July 2004.

[90] M. Hidalgo-Herrero. *Formal Semantics for a Parallel Functional Language*. PhD thesis, Universidad Complutense de Madrid, 2004. In spanish.

[91] M. Hidalgo-Herrero and Y. Ortega-Mallén. A distributed operational semantics for a parallel functional language. In *SFP'00 - Scottish Functional Programming Workshop*. Trends in Functional Programming, Vol. 2. Intellect, 1995.

[92] R. Hinze. Fortgeschrittene Algorithmen und Datenstrukturen in Haskell. Vorlesung Universität Bonn, 1998.

[93] R. Hinze. Polytypic Programming with Ease. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, number 1722 in LNCS. Springer-Verlag, November 1999.

[94] R. Hinze. A New Approach to Generic Functional Programming. In *Proceedings of the 27th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts, USA*. ACM Press, January 2000.

[95] R. Hinze. Generic Programs and Proofs. Habilitationsschrift, Universität Bonn, October 2000.

[96] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming, MPC Special Issue*, 43:129–159, 2002.

[97] R. Hinze and J. Jeuring. Generic Haskell: Applications. In *Lecture notes of the Summer School on Generic Programming*, number 2793 in LNCS. Springer-Verlag, 2003.

[98] R. Hinze and J. Jeuring. Generic Haskell: Practice and Theory. In *Lecture notes of the Summer School on Generic Programming*, number 2793 in LNCS. Springer-Verlag, 2003.

[99] R. Hinze, J. Jeuring, and A. Löh. Typeindexed Data Types. In *Proceedings of Sixth International Conference on the Mathematics of Program Construction*, number 2386 in LNCS, pages 148–174. Springer-Verlag, 2002.

[100] R. Hinze and A. Löh. Scrap your boilerplate revolutions. In *Proceedings of the 9th International Conference on the Mathematics of Program Construction (MPC)*, 2006.

[101] R. Hinze, A. Löh, and B. C. d. S. Oliveira. Scrap Your Boilerplate Reloaded. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'06), Fuji Susono, Japan*, 2006.

[102] Ralf Hinze. A generic programming extension for Haskell. Technical Report UU-CS-1999-28, Universiteit Utrecht, 1999. In Erik Meijer, editor, Proceedings of the 3rd Haskell Workshop, Paris, France September 1999.

[103] Ralf Hinze and Simon Peyton Jones. Derivable type classes. *Electronic Notes in Theoretical Computer Science, Elsevier*, 41(1), August 2001.

[104] P. Hruschka, C. Rupp, and G. Starke. *Agility kompakt*. Spektrum Akademischer Verlag, 2004.

[105] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[106] R.J.M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Oxford University Computing Laboratory, July 1983.

[107] Th. Ihringer. *Allgemeine Algebra*. Teubner-Studienbücher, 1993.

[108] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[109] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.

[110] C. Jay and P. Steckler. The functional imperative: shape! In *Proceedings of the 7th European Symposium on Programming (ESOP'98)*, number 1381 in LNCS, pages 139–153. Springer-Verlag, 1998.

[111] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129. Springer-Verlag, Berlin, 1996.

[112] J. Jeuring and P. Jansson. PolyP - a polytypic programming language extension. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 470–482, January 1997.

[113] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *LNCS 925, 1st International School on Advanced Functional Programming, Båstad, Sweden*, 1995.

[114] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993.

[115] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[116] Richard B. Kieburtz. Reactive Functional Programming. In *PROCOMET'98*, pages 263–284. Chapman and Hall, June 1998.

[117] David J. King, Jon G. Hall, and Philip W. Trinder. A strategic profiler for glasgow parallel haskell. In *Implementation of Functional Languages (IFL), London, UK*, pages 88–102, 1998.

[118] U. Klusik, R. Loogen, and S. Priebe. Controlling Parallelism and Data Distribution in Eden. In *Trends in Functional Programming (Selected papers of the Second Scottish Functional Programming Workshop)*, volume 2, pages 53–64. Intellect, 2000.

[119] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden: Low-Effort Parallel Programming. In M. Mohnen and P. Koopman, editors, *12th International Workshop, IFL 2000, Aachen*, LNCS 2011. Springer-Verlag, 2001.

[120] Ulrike Klusik, R. Peña, and C. Segura. Bypassing of Channels in Eden. In *SFP'99 - Scottish Functional Programming Workshop*, Trends in Functional Programming, pages 2–10. Intellect, 2000.

[121] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley, 2003.

[122] H. Kuchen. A skeleton library. In *Proceedings of Euro-Par 2002, LNCS 2400*, pages 620–629. Springer-Verlag, 2002.

[123] R. Lämmel and S. Peyton Jones. Scrap your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of ACM Sigplan Types in Language Design and Implementation (TLDI)*, 2003.

[124] R. Lämmel and S. Peyton Jones. Scrap more Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the International Conference on Functional Programming (ICFP'04)*, pages 244–245, 2004.

[125] R. Lämmel and S. Peyton Jones. Scrap your Boilerplate With Class: Extensible Generic Functions. In *Proceedings of the International Conference on Functional Programming (ICFP'05), Tallinn, Estonia*, pages 204–215, 2005.

[126] R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. Unpublished. Available from the author's web page., October 2002.

[127] R. Lämmel and J. Visser. Design Patterns for Functional Strategic Programming. In *Proceedings of third ACM Sigplan Workshop on Rule-based Programming*, 2002.

[128] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proceedings of the International Conference on Practical Aspects of Declarative Languages (PADL), Portland, Oregon, USA*, 2002.

[129] John Launchbury. Lazy Imperative Programming. Technical report, Computing Science Department, Glasgow University, December 1993.

[130] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI), Orlando*, June 1994.

[131] John Launchbury and Simon L. Peyton Jones. State in Haskell. Technical report, Kluwer Academic Publishers, November 1994.

[132] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[133] Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer: HaRe, and its API. In John Boyland and Grel Hedin, editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, April 2005. Published as Volume 141, Number 4 of Electronic Notes in Theoretical Computer Science.

[134] INMOS Limited. *Occam2 Reference Manual*. Prentice Hall, 1988.

[135] A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*. ACM Press, 2003.

[136] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, Netherlands, September 2004.

[137] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á. J. Rebón Portillo, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 16(3), 2003.

[138] H.-W. Loidl, P W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.

[139] Hans-Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, 1997. TR-1998-7.

[140] Hans-Wolfgang Loidl and Phil Trinder. Engineering Large Parallel Functional Programs. In *Selected papers from the 9th International Workshop on Implementation of Functional Languages*, volume 1467 of *Lecture Notes In Computer Science (LNCS)*, pages 178–197, 1997.

[141] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. *Patterns and Skeletons for Parallel and Distributed Computing*, chapter Parallelism Abstractions in Eden. In Rabhi and Gorlatch [176] , 2003.

[142] Rita Loogen. *Parallele Implementierung funktionaler Programmiersprachen*. PhD thesis, RWTH Aachen, January 1989. In german.

[143] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[144] Ian Lynagh. Template Haskell: A report from the field. Unpublished. Available from the author's web page., May 2003.

[145] Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. Unpublished. Available from the author's web page., May 2003.

[146] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[147] Jan-Willem Maessen. Eager Haskell: resource-bounded execution yields efficient iteration. In *2002 ACM SIGPLAN workshop on Haskell, Pittsburgh, Pennsylvania*, pages 38–50, 2002.

[148] Simon Marlow. Haddock, A Haskell Documentation Tool. In *Proceedings of the ACM SIGPLAN workshop on Haskell, Pittsburgh Pennsylvania, USA, ACM Press*, October 2002.

[149] R. Martínez and R. Peña. Building an Interface Between Eden and Maple: A Way of Parallelizing Computer Algebra Algorithms. In *IFL 2003, Edinburgh*, 2004.

[150] L. Meertens. Calculate polytypically! In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes In Computer Science*, pages 1–16, 1996.

[151] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In *Advanced Functional Programming (AFP 95)*, LNCS 925, pages 228–266. Springer-Verlag, 1995.

[152] Message Passing Interface Forum, University of Tennessee, Knoxville. *MPI: A Message-Passing Interface Standard*, May 1994.

[153] Message Passing Interface Forum, University of Tennessee, Knoxville. *MPI-2: Extensions to the Message-Passing Interface*, March 1997.

[154] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[155] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48(3):443–453, 1970.

[156] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van 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-Verlag, 1991.

[157] T. Nordin and S. L. Peyton Jones. Green card: a foreign-language interface for Haskell. In *Proceedings of the Haskell Workshop*, Amsterdam, Netherlands, June 1997.

[158] Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Proceedings of the International Workshop on the Implementation of Functional Languages*, pages 168–184, 2003.

[159] Ulf Norell and Patrik Jansson. Prototyping generic programming in template haskell. In *Proceedings of the 7th International Conference on the Mathematics of Program Construction (MPC)*, 2004.

[160] Oak Ridge National Laboratory. Parallel Virtual Machine, 2003. See: `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[161] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[162] Chris Okasaki. An Overview of Edison. In *Proceedings of the Haskell Workshop 2000*, 2000.

[163] Lawrence Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[164] S. Peyton Jones et al. Haskell 98: A Non-strict, Purely Functional Language, 2003. See: `http://www.haskell.org/onlinereport`.

[165] S. Peyton Jones et al. The Glorious Glasgow Haskell Compilation System, Version 6.4, 2005. Available at: `http://www.haskell.org/ghc`.

[166] S. Peyton Jones, M. Jones, and E. Meijer. Type Classes: Exploring the Design Space. In *Proceedings of the Haskell Workshop, Amsterdam, Netherlands*, 1997.

[167] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, 2001.

[168] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[169] R. Pointon, P. Trinder, and H.-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In *International Workshop on the Implementation of Functional Languages*, LNCS 2011. Springer-Verlag, 2000.

[170] Michael Poldner and Herbert Kuchen. On Implementing the Farm Skeleton. In *Proceedings of the 3rd International Workshop on High-Level Parallel Programming and Applications (HLPP), Warwick, UK*, 2005.

[171] S. Priebe. Concepts for the Communication in the Eden Runtime System. Master's thesis, Philipps-Universität Marburg, September 1998.

[172] Steffen Priebe. Preprocessing Eden with Template Haskell. In Robert Glück and Michael R. Lowry, editors, *Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2005.

[173] Steffen Priebe. Dynamic Task Generation and Transformation Within a Nestable Workpool Skeleton. In Wolfgang Nagel et al., editors, *EuroPar 2006 – Parallel Processing*, volume 4128 of *LNCS*, Dresden, Germany, 2006. Springer.

[174] Steffen Priebe. Towards Generic Rapid Prototyping of Parallel Functional Programs. Technical Report 53, Philipps-Universität Marburg, July 2006.

[175] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 2nd edition, 1994.

[176] F. A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2003.

[177] A. Reid. Template Greencard. In *Proceedings of 15th International Workshop on the Implementation of Functional Languages (IFL 2003)*, Edinburgh, September 2003.

[178] David Roundy. Darcs: a free, open source source code management system. Available at `http://darcs.net`, 2003.

[179] Winston Royce. Managing the Development of Large Software Systems. *IEEE WESCON*, 26:1–9, August 1970.

[180] F. Ruehr. Structural polymorphism. In R. Backhouse and T. Sheard, editors, Informal Proceedings Workshop on Generic Programming, Marstrand/Sweden, 1998.

[181] Dan Russell. *FAD: A Functional Analysis and Design Methodology*. PhD thesis, Computing Lab, University of Kent, Canterbury, UK, January 2001.

[182] Chris Ryder. *Software Measurement for Functional Programming*. PhD thesis, Computing Lab, University of Kent, Canterbury, UK, August 2004.

[183] P. Sadayappan, Y. L. C. Ling, and K. W. Olson. A restructurable VLSI robotics vector processor architecture for real-time control. *IEEE Transactions on Robotics and Automation*, 5(5):583–601, 1989.

[184] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *22nd ACM Symposium on Principles of Programming Languages, San Francisco, California*, January 1995.

[185] André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1995.

[186] S. Seefried, M. Chakravarty, and G. Keller. Optimising Embedded DSLs using Template Haskell. In *Third International Conference on Generative Programming and Component Engineering*, pages 186–205. Springer-Verlag, October 2004.

[187] Clara Segura Díaz. *Program Analysis in Parallel Functional Languages*. PhD thesis, Universidad Complutense de Madrid, Spain, 2001.

[188] Julian Seward. Towards a Strictness Analyser for Haskell: Putting Theory into Practice. Master's thesis, University of Manchester, 1991.

[189] T. Sheard and S. Peyton Jones. Template Meta-programming for Haskell. In *Haskell Workshop 2002*. ACM Press, October 2002.

[190] Tim Sheard. Languages of the future. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 116–119, 2004.

[191] Abraham Silberschatz and Peter Galvin. *Operating System Concepts*. Addison-Wesley, fifth edition, 1998.

[192] A. Simon and D. Coutts. Gtk2Hs: A GUI Library for Haskell based on Gtk. Available under `http://haskell.org/gtk2hs`, 2005.

[193] D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In J.S. Kowalik and L. Grandinetti, editors, *NATO ARW "Software for Parallel Computation"*, volume 106. Springer-Verlag NATO ASI, 1993.

[194] David B. Skillicorn and Domenico Talia, editors. *Programming Languages for Parallel Processing*. IEEE Computer Society Press, 1994.

[195] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[196] Guy Steele. Parallel Programming and Parallel Abstractions in Fortress. Invited talk at the Eighth International Symposium on Functional and Logic Programming (FLOPS), Fuji Susono, Japan, April 2006.

[197] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 2nd edition, 1992.

[198] Björn Struckmeier. Eden Trace Viewer: A Reimplementation in Haskell. Diploma thesis, Philipps-Universität Marburg, Germany, 2006. In german.

[199] Walid Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, USA, November 1999.

[200] Walid Taha et al. MetaOCaml: A compiled, type-safe, multi-stage programming language, 2006. See: `http://www.metaocaml.org`.

[201] Peter Thiemann. *Grundlagen der funktionalen Programmierung*. B. G. Teubner, Stuttgart, 1994.

[202] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.

[203] A. Tolmach et al. An External Representation for the GHC Core Language, September 2001. (Draft for GHC 5.02 documentation).

[204] Carmen Torrano and Clara Segura. Strictness analysis and let-to-case transformation using Template Haskell. In M. van Eekelen, editor, *6th Symposium on Trends in Functional Programming (TFP 2005)*, pages 429–442. Institute of Cybernetics, Tallinn, Estonia, 2005.

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

[206] P. W. Trinder, E. Barry, Jr., M. K. Davis, K. Hammond, S. B. Junaidu, U. Klusik, H.-W. Loidl, and S. L. Peyton Jones. GPH: An Architecture-Independent Functional Language. *IEEE Transactions on Software Engineering*, 1999.

[207] 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*. Universität Kiel, 1996.

[208] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, 1990.

[209] T. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, 1998.

[210] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98), Baltimore, USA*, volume 34/1 of *ACM SIGPLAN Notices*, pages 13–26. ACM Press, 1999.

[211] Eelco Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. In *Rewriting Techniques and Applications (RTA'01), Utrecht, The Netherlands*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, May 2001.

[212] Philip Wadler. *Strictness analysis on non-flat domains (by abstract interpretation over finite domains)*, pages 266–275. Ellis Horwood, 1987.

[213] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, volume 2, pages 461–493, 1992.

[214] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*. Austin, Texas, January 1989.

[215] K. Wansbrough. Macros and Preprocessing in Haskell. Unpublished, available from the author's webpage, 1999.

[216] N. Winstanley. A Type-Sensitive Preprocessor for Haskell. In *Glasgow Workshop on Functional Programming, Ullapool, Scotland*, 1997.

[217] A. Zomaya. *Parallel and Distributed Computing Handbook*. McGraw-Hill Series on Computer Engineering, 1996.

Curriculum Vitae

Steffen Priebe
(verheiratet, eine Tochter)

10. Juni 1972	geboren in Marburg
September 1991	Abitur am Gymnasium Martin-Luther-Schule in Marburg
Okt. 1991 - Okt. 1992	Zivildienst bei der Blista EHG in Marburg
Okt. 1992 - Sep. 1998	Studium der Informatik an der Philipps-Universität Marburg (“Mit Auszeichnung bestanden”)
Okt. 1998 - Jan. 2004	Wissenschaftlicher Mitarbeiter am Fachbereich Mathematik u. Informatik der Philipps-Universität Marburg
Jun. 2001 - Mai 2003	Unterbrechung der Dissertation wegen schwerer Erkrankung
Apr. 2004	Aufnahme als Stipendiat in die Promotionsförderung des Evangelischen Studienwerkes Villigst e.V.
	Wiederaufnahme der Dissertation