A Persistent Functional Language for Concurrent Transaction Processing

Lesley Wevers

Master's Thesis

Department of Computer Science
University of Twente

Graduation Committee:
dr. Marieke Huisman
dr.ir. Ander de Keijzer
prof.dr. Jaco van de Pol

August 24, 2012

Abstract

In this thesis we investigate the construction of transaction processing systems using functional programming languages. The traditional method for the construction of a transaction processing system is to use a database management system (DBMS) together with a general purpose programming language (GPPL). However working with a DBMS from a GPPL is difficult due to the impedance mismatch, and this model limits the potential concurrency of the system as a whole. We have developed a prototype persistent functional language for transaction processing that solves these problems.

In our language, states are a set of bindings from values to expressions. Transactions may evaluate expressions in the context of the current state, and they may update the bindings in the state. In our approach, a DBMS is implemented in our language using bulk data structures. A transaction processing application can be implemented within the same system, thus resolving the impedance mismatch. A domain specific interface for the transaction processing application can be created using stored transactions. Additionally, our systems allows ad-hoc querying and manipulation of the data through an interactive interpreter.

Our language model can be implemented efficiently through graph reduction. Concurrency can be introduced through lazy reduction of states, which allows higher levels of concurrency than existing concurrency control methods. We have implemented a graph reducer based on template instantiation, which has been adapted to allow bindings to be created dynamically by resolving references to supercombinators statically such that unused bindings can be garbage collected automatically. We have implemented a transaction manager for our language model that allows the concurrent execution of transactions. Additionally, we introduce a novel approach to parallel graph reduction, where we distribute work among reduction threads by randomising the reduction order of strict function arguments, and by ensuring that reduction results are correctly shared between reduction threads.

Further, we investigate methods for storing states in persistent memory. One method is based on snapshotting the state of the system, allowing checkpointing of ongoing computations. Another method is based on log-structured storage, and allows storage of large states with low recovery times. We combine both of these approaches to allow checkpointing of ongoing computations, storing states larger than main-memory, and supporting low recovery times. In all of these approaches we use journaling as a method to ensure durability of transactions. For our prototype we have implemented journaling together with a simplified version of snapshotting that does not support checkpointing of ongoing computations.

Finally, experiments with our graph reducer show nearly ideal relative speedup in two example programs, without the need to explicitly annotate the programs for parallelism. Additionally, our experiments show that lazy reduction of states may lead to long chains of suspensions and memory leaks in the state. We propose a solution where we force the evaluation of states, and limit the number of active transactions.

Acknowledgements

First, I would like to thank my supervisors: Marieke Huisman, Ander de Keijzer and Jaco van de Pol. They have provided me the opportunity to define my own master's project, as well as providing me vital feedback while writing this thesis. Marieke was my main advisor, with whom I had a SCRUM meeting almost every day. I would especially like to thank Ander for offering up his free time to help me with this thesis after he left the university.

I would also especially like to thank Stefan Blom who always took the time to discuss problems, and who provided many technical suggestions of which some led to the results in this thesis, most notably the combined approach to storing states in persistent memory. Also, I would like to thank Elmer Lastdrager with whom I had a daily coffee break, and who provided me with many helpful suggestions.

From my group, I would like to thank Alfons Laarman and Tom van Dijk for taking the time and helping me with the system on which I have run the experiments, and providing insight into parallel algorithms. Also, I would like to thank Maarten de Mol for helping me on the topic of functional programming.

Also, I would like to thank my fellow final year students with some of whom I have shared the office: Freark van den Berg, Harold Bruintjes, Ronald Burgman, Gerjan Stokkink, Paul Stapersma and Vincent de Bruijn. They have provided me with a nice environment at the university, although sometimes a bit too 'gezellig'. I would also like to thank the rest of the FMT group for nice conversations during lunch as well as the occasional discussion.

Last but not least, I would like to thank my mother Inge, my steph-mother Sientje, my brother Lennart and my sister Laurie for supporting me in writing this thesis and providing a nice environment at home.

Contents

1. Introduction

Where transaction processing systems could once only be found in the realm of large organisations, the decreasing cost of computing resources and the advent of the internet have made transaction processing systems an integral part of many small organisations and almost every website. Typical examples of transaction processing systems include banking systems, ticket reservation systems and inventory management systems. A transaction processing system often manages all the data of an organisation. Data that has to be available instantly, sometimes to many thousands of simultaneous users, while providing the illusion that each user has exclusive access to the data. It is also crucial that the data is kept safe from system failures, outside attackers, as well as programming mistakes. All of these requirements make the construction of a transaction processing system a challenging task.

In this thesis, we investigate the use of functional programming languages for the construction of transaction processing systems. It has already been known for some time that functional languages provide an interesting basis for the implementation, querying and manipulation of databases [26, 28, 38], which are an essential part of a transaction processing system. Interesting properties of functional programs are that they can be executed lazily and in parallel. When a program is executed *lazily*, only those parts of program are executed that are necessary to produce its result. This provides a basis for concurrent execution of transactions written in a functional programming language, as we only have to compute the modifications to those parts of the database that a transaction requests, thereby enabling fast response times. Laziness also ensures that we only access the minimal part of the data that is necessary for the execution of a transaction, thereby minimising access to slow persistent storage media such as hard-disk drives. Another interesting property of functional programs, is that they are inherently *parallel*, allowing execution of transactions in parallel without explicitly introducing parallelism. Finally, functional programs are known to be relatively easy to reason about, providing a basis for verifying correctness of transaction processing systems.

1.1. Traditional Transaction Processing Systems

Before we describe our approach, we outline a common approach to the construction of transaction processing systems so that we can contrast this to our approach.

A *database management system* (DBMS) is a very generic kind of transaction processing system [17]. The main feature of a DBMS is that it is optimised to handle a very large amount of data stored in persistent memory. A DBMS typically provides:

- A *data model* that determines how databases are structured, and which also defines the basic operations that can be performed on databases. Examples of data models include the relational model [13] and the XML data model [17].
- A *programming interface* for the interrogating and manipulation of databases. This is usually a high-level declarative language, allowing user to perform *ad-hoc* queries, leaving it up to the DBMS to determine how to execute queries efficiently. The programming interface is usually explicitly designed to work with the data model provided by the DBMS. For example, in relational databases a common programming interface is SQL, while for XML databases XQuery and XPath are commonly used.

While a DBMS could in principle be used as a complete transaction processing system in itself, end users do not typically work directly on the DBMS for to two main reasons: the programming interface is not user friendly for the end users of the system, and it poses a security risk. To solve these problems, a transaction processing systems usually also consists of an *application* that provides an interface to the users of the system, such as a website or a physical device such as an ATM [23]. Figure 1.1 shows a an overview of such an architecture.

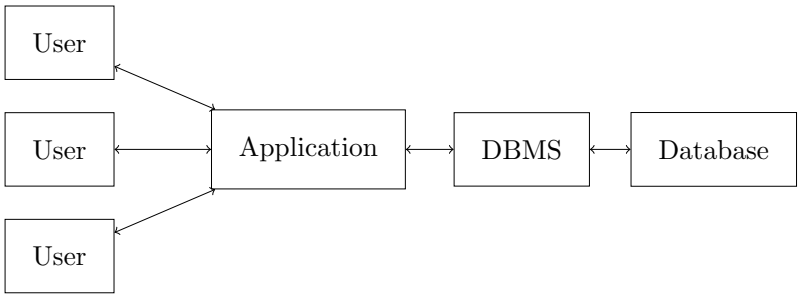


Figure 1.1.: Architecture of a simple transaction processing system.

The application translates transaction requests from the user into database programs that are sent to the DBMS, and translates responses from the DBMS into responses suitable for end users. Additionally, the application can enforce additional domain specific security rules that can not be enforced by the DBMS itself.

1.2. Problems in the Traditional Model

Constructing transaction processing in the traditional model presents some difficulties in practice. The application is typically a separate process from the DBMS, which may even run on a different machine than the DBMS. This means that the system is actually a distributed system, as components can fail independently. Furthermore, applications are typically created using a general purpose programming language (GPPL), which data model, type system and computational model is usually very different from those

provided by the DBMS. This divide between the DBMS and the application creates several problems, as described in the remainder of this section.

First, queries on the database are usually built dynamically from inside the GPPL by composing strings and values. This creates the possibility of *command injection attacks* [36], as well as making it difficult to validate the correctness of an application because regular type checking can not be applied.

Second, the mapping between the data model of the GPPL and the data model of the DBMS is often complex and unnatural, a problem known as the *impedance mismatch* [17]. A programmer is required to explicitly map concepts from the DBMS to concepts in the programming language, and vice versa. Also, a database usually supports only a fixed set of data types, where an application may need additional data types. Creating a mapping between data types requires a lot of effort from the programmer, and is prone to error.

Third, because the system is distributed, the application programmer has to take care that DBMS failure is handled correctly. This complicates the implementation of the application as failure may occur at any point in the program where it communicates with the DBMS. Additionally, as the application and the database are different processes, they may evolve separately. Even if the correctness of the application has been verified, the schema of the database may change, which could lead to incorrect behaviour if the application is not updated.

Fourth, the communication between the application and the DBMS usually takes place over a network, which may incur a high overhead in the execution time of transactions. This is especially the case if the execution of a transaction requires a lot of interaction with the DBMS. Also, communication between an application and a DBMS is often performed sequentially. This limits the potential to execute transactions in parallel. Additionally, as the DBMS does not know about the future actions of a user updating the database, the DBMS is unable to perform certain optimisations to the concurrent execution of transactions, limiting the performance of a DBMS while executing transactions concurrently.

Finally, concurrent execution of transactions may lead to concurrency conflicts. In order to resolve or avoid concurrency conflicts, a DBMS may respectively abort transactions or block access to parts of the state while another transaction is in progress [17]. The programmer has to ensure that transaction aborts are always handled correctly. Additionally, a DBMS may repeatedly abort transactions if there is high contention for data, or if a transactions affect a large part of the database. This may leave a user unable to execute a transaction, a problem known as *starvation*. If a DBMS instead chooses to avoid concurrency conflicts by blocking access to the state, slow users or large updates may prevent access to the state for a long period of time.

1.3. Persistent Functional Languages

An approach to solve many of the problems as described in the previous section is to integrate a GPPL with the features of a DBMS, as to close the divide between them. Such a system can either be seen as a *persistent programming language* [4], or a DBMS with an integrated programming language. For the purpose of this thesis we use the term persistent programming language, which can be seen as a programming language that transparently manages the storage of its state in persistent memory. Persistent programming languages solve many of the problems as discussed in the previous section: queries can be written in the same language as that is used for the construction of the application, the application and the DBMS are under the same type system, the system is not distributed, and there is no network communication overhead between the DBMS and the application.

Many attempts have already been made towards the development of persistent programming languages [1, 2, 4, 14, 25], but some of the problems as discussed in the previous section still remain. We observe that almost all of the existing systems are based on the use of imperative programming languages to define transactions. Using functional programming languages can solve some of the remaining problems.

One problem with the imperative programming model is that it is very different from the declarative model of many database query languages. In an imperative language, you not only have to tell the system what you want, but also how the system has to do it. This makes imperative languages difficult to use for querying and manipulating databases, and it is difficult to optimise transactions in an imperative language. In contrast, functional languages are quite close to the declarative model of database query languages. Research has been done that shows that list comprehensions in functional languages are relationally complete, and that functional transactions can be optimised similarly to optimising declarative database queries [38]. This makes functional languages well suited for querying databases.

Another problem when using imperative languages for defining transactions is that they are not well suited for concurrent execution [7]. One of the reasons for this is that imperative transactions must be executed sequentially. In contrast, there is flexibility in the execution order of a functional program, as any execution order produces the same result. This allows concurrency control through lazy evaluation of transactions, allowing higher levels of concurrency than possible using imperative languages. Furthermore, the flexibility in execution order also allows parallel execution of independent parts of a transaction.

1.4. Our Approach

Our approach to the construction of a persistent functional language is to integrate a functional programming language with transaction processing capabilities. A functional transaction executes in the context of a state, and produces a new state together with

an observable result. In order to provide persistence, we transparently store the states produced by transactions in persistent memory.

In our approach, the system itself does not provide a data model. Instead, a DBMS with a corresponding data model is implemented in our language. This makes our system very flexible, as any kind of data model can be implemented. A standard library of DBMS implementations can be provided with the language such that a user does not have to implement a DBMS itself. The systems ensures that concurrent operations on the database are executed correctly, and storage of states in persistent memory is handled automatically.

Transactions are written in the same language that is used to implement the DBMS, thus solving the impedance mismatch. Furthermore our language allows the definition of stored transactions to provide a method for the construction of the application part of a transaction processing systems. Figure 1.2 shows the architecture of a transaction processing system in our approach.

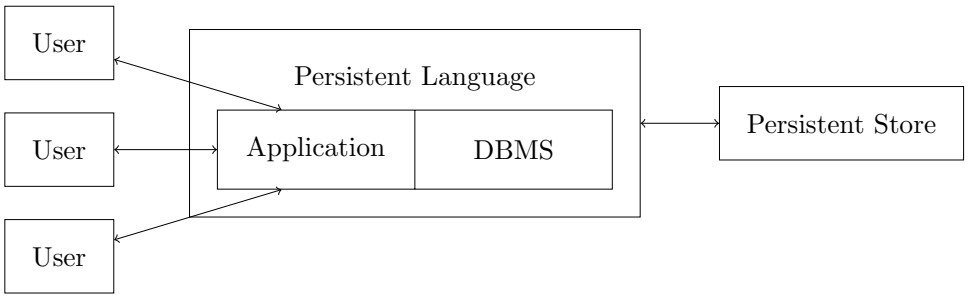


Figure 1.2.: Architecture in our approach.

In contrast to the traditional approach, in our approach a transaction is sent in its entirety to the system, instead of using a sequential interface to issue individual queries. Because of this, all operations on the database are known when the transaction starts, allowing the execution of independent parts of the transaction in parallel. This also means that slow clients can not affect the performance of the system, as there is no interaction during the execution of a transaction. Further, transactions are guaranteed to commit, so there is no explicit management of transaction commits and aborts. A potential disadvantage of our approach is that the system does not support interactive transactions, however we have ideas to solve this issue, which is discussed briefly in the future work section.

Concurrent execution of transactions is achieved through lazy evaluation of states [28,38], which provides concurrency control by means of data dependency between transactions, without the risk of deadlock. Intuitively, concurrency conflicts are prevented by blocking access to parts of the state in a very fine grained manner, where parts of the state can be unblocked by evaluating those parts first. This allows laziness to provide higher levels of concurrency than is possible in a traditional DBMS.

1.5. Early Work and Goals

Before we discuss our contributions, we shortly review earlier work that has been done in the area of functional persistent languages.

In 1985, Nikhil proposed the use of functional languages for the implementation, querying and updating of functional databases [28]. In particular, Nikhil proposed that a functional database can be viewed as an environment that maps identifiers to expressions. A transaction may evaluate an expression in the context of an environment to query the environment, and a transaction may update the database by replacing the environment with a new environment. Furthermore, Nikhil sketched an approach to implement what he calls a functional database programming language.

In 1989, Trinder [38] also explored the use of functional languages to construct functional databases. He defined a model where transactions are functions that take the current state of the database, and produce a new state of the database. A transaction manager takes a stream of such transactions, and executes them sequentially, producing a stream of results. As the state, Trinder uses bulk data structures such as binary trees to implement functional databases. Trinder showed that lazy evaluation of the states produced by transactions allows concurrent execution of transactions. Furthermore, he identified limitations to concurrency, as well as identifying approaches to overcome some of these limitations.

Existing implementation of persistent functional languages include STAPLE by McNally [26], and AGNA by Nikhil [29]. Both of these approaches are based on the model by Nikhil. However, a limitation of both approaches is that they do not support the concurrent execution of transactions. STAPLE supports persistence through the use of a generic persistent object system, while Agna supports persistence through paging but does not satisfy the durability requirements for transaction processing.

The main goal of this thesis is to develop a persistent functional language that supports concurrent transaction processing, and we want to investigate methods for persistence that are optimised for functional languages. As a starting point for our language we use the model as described by Nikhil and Trinder [38]. Concretely, our goals are to develop:

1. A functional language that can be used for transaction processing.

2. Methods for the concurrent execution of transactions.

3. Methods for the efficient storage of functional states in persistent memory.

To validate the feasibility of our solutions to these goals, we implement a prototype of our language in Java as a proof of concept, and with which we perform an experimental analysis.

1.6. Contributions

Concretely, this thesis describes the following contributions:

- a language for the definition of transactions in a functional language, including stored transactions for the construction of applications;
- a prototype implementation of our language that supports concurrent execution of transactions and storing states in persistent memory;
- a graph reducer based on template instantiation that has been adapted for transaction processing;
- a new method for load balancing in parallel graph reduction, based on sharing results between reduction threads, and randomising their reduction order;
- a discussion of different methods for storing functional states in persistent memory, allowing both the storage of suspended computations, as well as supporting states that are larger than main-memory;
- forcing the evaluation of transactions as a solution to space leaks that show up in the theoretical model due to lazy reduction of states; and
- an experimental evaluation of our prototype.

1.7. Thesis Outline

In the first part of the thesis, we present a study on the background of functional transaction processing. First, we review functional programming in Chapter 2. Then in Chapter 3, we provide an overview of functional transaction processing and we discuss a model for transactional functional languages.

In the second part of the thesis, we describe our contributions. We begin by describing our language for defining functional transactions in Chapter 4. Next, in Chapter 5 we describe a graph reducer that has been adapted to allow dynamic creation of bindings. In Chapter 6, we present our load balancing method for parallel graph reduction. In Chapter 7, we describe the implementation of a concurrent transaction manager for our language, and we forcing the evaluation of states. Finally, in Chapter 8 we describe methods for storing states in persistent memory.

In the last part of the thesis, we evaluate our contributions. We start out by evaluating our prototype implementation by means of experiments in Chapter 9. Then, in Chapter 10 we review related work on persistent languages, functional transaction processing, and parallel graph reduction. Finally, in Chapter 11 we present our conclusions and provide directions for future work.

Part I.

Background

2. Functional Programming

In this chapter we review functional programming, which provides an alternative model of computation to the commonly used imperative programming model. Computation in the imperative model is performed as a side-effect of the sequential execution of instructions. In contrast, functional programming models computation in terms of the evaluation of expressions. A functional program is essentially an expression that is reduced until a non-reducible expression is obtained, which is the result of the program. Functional programming allows a form of reasoning that is similar to reasoning in mathematics. Additionally, in contrast to the sequential nature of imperative programs, functional programs are naturally concurrent, and can be executed in parallel without affecting the result of a program.

In this chapter we first discuss the lambda-calculus, which provides the theoretical basis of functional programming. Next, we discuss how functional programming languages extend the lambda-calculus. After that we discuss the reduction order in the execution of functional programs. And finally, we discuss graph reduction as a method to implement lazy functional languages efficiently.

2.1. The Lambda Calculus

At the basis of functional programming is the λ-calculus [5], which is a model of computation introduced in 1936 by Church [12]. Computation in the λ-calculus is not based on the execution of instructions, but on the reduction of expressions in the λ-calculus. In this model, a computational problem can be encoded as a λ-calculus expression, which when fully reduced produces the result to the problem. In this section we provide a quick overview of the λ-calculus, and we define some of the terminology that we use in this thesis.

An expression in the λ-calculus consists of three basic syntactic elements:

- *variables*: x, y, z, ...;
- *lambda abstractions* of the form $\lambda x.E$; and
- *applications* of the form $(E_1\ E_2)$.

Where E, E_1 and E_2 are expressions in the λ-calculus. Additional parenthesis may be inserted into a λ-calculus expression to improve readability, and parenthesis may be omitted if the meaning of an expression is obvious.

Some examples of expressions in the $\lambda - calculus$ are:

$$y$$
$$(\lambda x.x)$$
$$((\lambda x.x)\ y)$$
$$(\lambda x.(x\ x)\ \lambda x.(x\ x))$$

Bound Variables and Free Variables

A lambda abstraction of the form $\lambda x.E$ is said to *bind* all occurrences of x in E. A variable x is *bound* if x is part of the body of a lambda-abstraction that binds x. A variable x is *free* if it is not part of the body of a lambda-abstraction that binds x. For example, in the expression $\lambda x.(x\ y)$, x is a bound variable, and y is a free variable. An expression is said to be *closed* if it does not contain free variables.

Reduction

Reduction of a λ-calculus expression is performed by means of β-reduction, which is defines as a rewrite rule $((\lambda x.E)\ s) \rightarrow E[x := s]$, where $E[x := s]$ is a *capture-avoiding substitution* that substitutes all free occurrences of x in E by s. For example, the expression $((\lambda x.(x\ x))\ y)$ is reduced as follows:

$$((\lambda x.(x\ x))\ y) \rightarrow (x\ x)[x := y] \rightarrow (y\ y)$$

Under β-reduction, a lambda-abstraction $\lambda x.E$ can intuitively be seen as a definition of an anonymous function of a single variable x with a *body* E. An application $(E_1\ E_2)$ can intuitively be seen as a call or invocation of a function E_1 with the parameter E_2.

An expression of the form $((\lambda x.E)\ s)$ is called a *reducible expression*, or *redex* for short. If an expression does not contain any redexes, it is said to be in *normal form*. For example, $(y\ y)$ is in normal form because it does not contain a redex, but $(y\ ((\lambda x.(x\ x))\ y))$ is not in normal form as is contains the redex $((\lambda x.(x\ x))\ y)$.

An expression can be reduced to normal form by repeatedly rewriting it using β-reduction. However, not every expression has a normal form, for example $(\lambda x.(x\ x))(\lambda x.(x\ x))$ reduces to itself, so its reduction does not terminate. However, if there exists a terminating reduction sequence, then the normal form of an expression is *unique*.

2.2. Functional Programming

While the λ-calculus is computationally complete, it is not very practical. Functional programming is a programming model that is built upon the λ-calculus [33], which solves this problem. A large body of work exists on functional programming, and many implementations are available such as Haskell [31], Miranda [39], Clean [11], Lisp [21], and many others. In this section we discuss how a functional programming language differs from the λ-calculus.

Functions

Encoding a program into a single λ-calculus expression is quite cumbersome. In functional programming languages, this problem is solved by allowing the definition of *functions*, which are named expressions that may refer to one another by their name. Usually there is one *main function* which, when reduced to normal form, produces the output of the program. During the execution of a functional program, references to functions are resolved by substituting them with their corresponding expression in the program. For example, consider the following program:

$$\begin{aligned} \mathit{incr} &= \lambda x.(x+1) \\ \mathit{main} &= (\mathit{incr}\ 7) \end{aligned}$$

The execution of this program consists of reducing the main expression $(\mathit{incr}\ 7)$. In order to do this, we first have to resolve the reference *incr*, obtaining the redex $((\lambda x.x+1)\ 7)$ which can then be reduced to 8.

Additionally, functional programming languages usually allow functions to be written in the more traditional form $f(x_1, \ldots, x_n) = E$, which can be translated to an expression $f = \lambda x_1\ .\ \ldots\ \lambda x_n\ .\ E$.

Data Constructor Functions

Functional programming languages also commonly provide methods to construct complex data types through data constructor functions. In contrast to regular functions, data constructor functions can not be reduced. For example, lists can be represented recursively using two data constructor functions, usually named *Cons* and *Nil*, where *Cons* represents a list element followed by a list, and *Nil* represents an empty list. To encode the list $[1, 2, 3]$, we can write:

$$\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))$$

In order to work with constructor functions, functional programming languages usually provide case distinction expressions. Consider the following example:

$$\begin{aligned} \mathit{length}(x) &= \mathit{case}\ x\ \mathit{of} \\ &\quad\ \mathit{Nil} \to 0 \\ &\quad\ \mathit{Cons}\ x\ \mathit{xs} \to 1 + \mathit{length}\ \mathit{xs} \end{aligned}$$

This function computes the length of a list x recursively. If the normal form of x is of the form Nil, then the case expression reduces to 0, and if the normal form of x is of the form $\mathit{Cons}\ x\ \mathit{xs}$, then the case expression reduces to $1 + \mathit{length}\ \mathit{xs}$.

Primitive Data Types

Encoding basic data types such as integers is not very practical in the λ-calculus, as well as being very inefficient in practice. For this reason most functional programming languages allow the use of primitive data types and primitive functions, such as number types and arithmetic functions. We have already seen the use of primitive data types in the examples above.

2.3. Lazy and Parallel Evaluation

At any time during the reduction of a functional expression, there may be multiple redexes that can be reduced. Typical programming languages evaluate the arguments of a procedure before invoking the procedure. Additionally, purely functional programming languages allow lazy evaluation of function arguments, delaying the evaluation of an argument until they are actually needed. For example, consider the following function:

$$f(x\ y) = \mathit{if}\ x > 0\ \mathit{then}\ x\ \mathit{else}\ y$$

If we reduce $f(4\ (3 \times 3))$ under eager evaluation we obtain the following reduction sequence:

$$f(4\ (3 \times 3)) \to f(4\ 9) \to \mathit{if}\ 4 > 0\ \mathit{then}\ 4\ \mathit{else}\ 9 \to 4$$

Under lazy evaluation we obtain the following reduction sequence instead:

$$f(4\ (3 \times 3)) \to \mathit{if}\ 4 > 0\ \mathit{then}\ 4\ \mathit{else}\ (3 \times 3) \to 4$$

We see that under lazy evaluation, (3×3) is not evaluated, as it is not needed to produce the normal form of the expression. Lazy evaluation is very important for our functional transaction processing language, as it allows concurrency by reducing states lazily, as discussed in Chapter 3.

Additionally, lazy evaluation allows the definition of control flow structures such as if-expressions as functions as opposed to special language constructs as is common in eagerly evaluated languages. Additionally, lazy evaluation has the property that if there exists a reduction order that terminates, then lazy evaluation terminates. In practice this allows the use of recursively defined infinite data structures in a functional program, such as the Fibonacci sequence, while still guaranteeing that the program terminates.

In order for lazy evaluation to be correct, functions must be pure. A *pure function* is a function that is deterministic and is side-effect free. A function is *deterministic* if given the same input, it always produces the same output. A function is *side-effect free* if it does not depend on or affect the environment in which the program is executed. For example a function that prints a string to the console is deterministic, but it is not side-effect free.

An important consequence of purity is that reduction of expressions that only contain pure functions results in a unique normal form. This means that during reduction any *reduction order* may be chosen, and we are guaranteed to obtain the same result. Eager evaluation corresponds to reducing the *innermost* redexes in an expression first, and lazy evaluation corresponds to reducing the *outermost* redexes in an expression first.

To precisely define the reduction order chosen by lazy evaluation, we use the notion of weak head normal form [33]. We say that an expression E is in *weak head normal form* if E is a lambda-abstraction, a data constructor function, or a primitive data type. In lazy evaluation, we always reduce toward weak head normal form. In Chapter 5 we discuss how this is done in our prototype implementation.

Another implication of purity is that sub-expressions may be reduced in parallel, while the final result remains deterministic, which means that the functional programming model is naturally parallel. This is in contrast to the imperative programming model, where parallelism generally has to be introduced explicitly, and where it may lead to non-deterministic results. However, in order to take advantage of parallel reduction, a program has to make sure there are always multiple sub-expressions that can be reduced at any one time.

2.4. Graph Reduction

Graph reduction is a method for the efficient implementation of lazy functional programming languages. To illustrate why graph reduction is needed, consider that we have a function $f(x) = x + x$, and we want to know the normal form of $f(2 \times 3)$. Under lazy evaluation we obtain the reduction:

$$f(2 \times 2) \to 2 \times 3 + 2 \times 3 \to 6 + 2 \times 3 \to 6 + 6 \to 12$$

Note that the argument 2×3 is duplicated when instantiating the function body of f, requiring us to evaluate the argument twice. Under eager evaluation this does not happen, as shown in the reduction:

$$f(2 \times 3) \to f(6) \to 6 + 6 \to 8$$

In graph reduction, expressions are represented by graphs, where nodes represent the syntactic elements of an expression, and edges represent a sub-expression relation between these expressions. Compared to representing expressions syntactically, a graph representation allows the sharing of sub-expressions. Graph reduction allows the efficient implementation of lazy reduction, because instead of duplicating arguments as in the example, it allows sharing the argument.

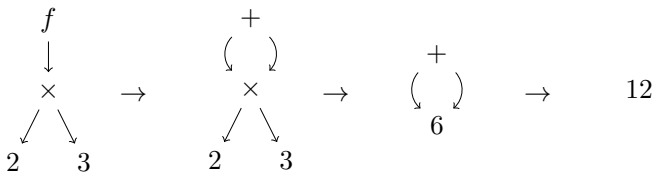


Figure 2.1.: Graph reduction of $f(2 \times 3)$.

Figure 2.1 shows how reduction of $f(2 \times 3)$ is performed using graph reduction. On the left side we see $f(2 \times 3)$ in graphical form. When we reduce this graph, we see that the sub-graph for 2×3 is shared when the body of f is instantiated. Sharing of a computation result is performed by *overwriting the root* of a reducible expression. Similarly, graph reduction allows sharing of data in order to conserve memory, as can be seen in the example by sharing the intermediate result 6.

2.5. Conclusions

In this chapter we have provided a quick review of functional languages, and the properties that are relevant for this thesis. In particular, we have seen:

- The lambda-calculus provides the basis of functional programming languages.
- Functional programming languages extend the lambda calculus with structured data, primitive data types and primitive functions, as well as allowing programs to be defined as multiple expressions that may refer to one another.
- Purity provides flexibility in the reduction order of functional programs, allowing lazy and parallel reduction.
- Graph reduction provides a method for the efficient implementation of lazy functional languages.

In the Chapter 3 we discuss how functional programming languages can be used for transaction processing, and in Chapter 5 we discuss our implementation of graph reduction.

3. Functional Transaction Processing

In this chapter we discuss the theoretical background of using functional languages for transaction processing . First, we discuss transaction processing in general. Next, we review the work by Trinder [38] as a model for functional transaction processing. After that, we show that this model can be implemented efficiently using graph reduction. Next, we show that this model allows concurrent execution of transactions by lazy reduction of states. Finally we discuss a model for transactional functional languages, based on the work by Nikhil [28].

3.1. Transaction Processing

In this section we first discuss the general concepts of transaction processing and transaction processing systems.

A *transaction* is a collection of operations on a state that provides guarantees about its execution as a whole [18]. Transactions are used in data management systems such as databases, file systems and version control systems, to ensure that operations on the data are executed correctly. Most such systems guarantee four correctness properties for the execution of transactions, known as the *ACID* properties [19]:

Atomicity: Either all operations in a transaction are executed, or none at all.

Consistency: Consistency rules are not violated in the states between the execution of transactions.

Isolation: The result of transactions executing in parallel is the same as the result for some sequential executions of the transaction.

Durability: Once a transaction has been committed, its effects must persist even in the case of system failure.

A transaction can be terminated by either *committing*, making its effects persistent, or *aborting*, undoing its effects A transaction may be aborted due to a violation of any of the ACID properties, on request by the initiator of the transaction, or due to connection or system failures during the execution of the transaction.

We can now formalise the property of isolation in terms of serialisability and recoverability:

Serialisability: A concurrent execution of a set of transactions producing some final state is *serialisable* if some sequential execution of the transactions produces that same final state.

Recoverability: Committed transactions may not have read data that is written by aborted transactions. This implies that, as long as a transaction t has not been committed, all transactions that have read changes by t can not commit until t commits, and if t chooses to abort, all transactions that have read changes by t must also abort.

A *transaction processing system* is a system that manages the concurrent execution of transactions by multiple users on a single state. A typical example of a transaction processing system is a banking system, where the state consists of bank accounts, and where transaction types include transferring funds between accounts, depositing funds, withdrawing funds, and adding interest.

In the literature, many examples of transaction processing systems can be found, ranging from simple single-computer systems, to very complex multi-server systems as can be found in large organisations [23]. This thesis only considers transaction processing in *shared memory systems*, where all state resides on one machine. Transaction processing can also be done in distributed systems, but this is outside the scope of this thesis.

Ensuring that the ACID properties hold results in two main challenges for transaction processing system implementations:

- In order to minimise execution times of individual transactions, the system has to be able to execute transactions concurrently, i.e. it must allow the execution of transactions to overlap in time. In order to ensure that serialisability and recoverability are not violated, a transaction processing system typically applies *concurrency control* techniques.

- The system has to maintain the ACID properties even in the case of a system failure. Writes to persistent storage may only be partially complete at the time of failure, which may lead to inconsistent states or transactions not being executed atomically. A transaction processing system usually solves this by *journaling* transactions before executing them, and by executing a *recovery* procedure after a system crash that restores the state using information from the logs. This is discussed in more detail in Chapter 8

In the remainder of this chapter, we discuss how functional languages can be used to construct transaction processing systems.

3.2. A Model for Functional Transaction Processing

In this section we discuss how functional languages can be used for the construction of transaction processing systems, by reviewing the work of Trinder [38].

A *transaction function* is a function of type `State` $\rightarrow$ `State` $\times$ `Result` that takes a state and produces a new state together with an *observable result*. Figure 3.1 shows a pictorial representation of a transaction function.

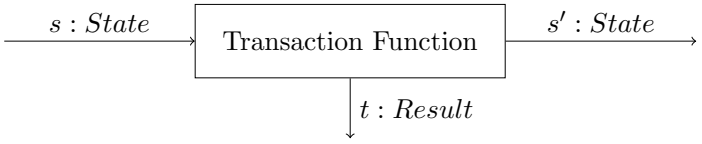


Figure 3.1.: A transaction function.

As an example, consider that `State` = `Integer` $\times$ `Integer` and `Result` = `String`. We could define the following transaction functions in a functional language:

```
1 get_a (a, b) = ((a, b), show a)
2 get_b (a, b) = ((a, b), show b)
3 swap (a, b) = ((b, a), "")
```

Here we assume that the `show` function converts its argument into a string. The `get_a` and `get_b` state transformers respectively produce as their observable result the first and second integer from the state, and return the state as it is. The `swap` state transformer swaps the two integers in the state, and returns an empty string as its observable result.

A functional transaction processing system can be constructed using a *transaction manager function* of type $State \times [Transaction] \rightarrow [Result]$ that takes an initial state and a stream of transactions, and produces a stream of results. A simple implementation of a transaction manager function in a functional programming language is the following:

```
1 tm : State × [Transaction] → [Result]
2 tm state [tx:txs] = result:(tm new_state txs) where
3   (new_state, result) = tx(state)
```

That is, given a state `state`, we take a transaction `tx` from the stream of transactions `txs`. We then execute the transaction on the state by invoking `tx(state)` to obtain a new state `new_state` and a result `result`. We concatenate the result to the result stream and produce the rest of the result stream by recursively invoking the `tm` function to process the rest of the transaction stream on the new state.

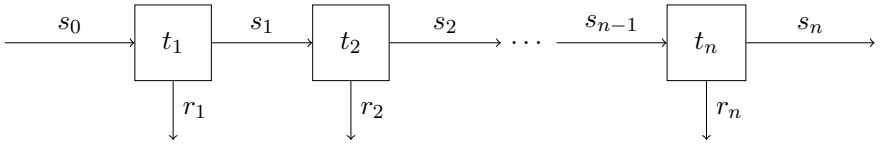


Figure 3.2.: Executing a stream of transactions.

Figure 3.2 shows how the `tm` function works for the execution of a stream of transaction functions t_1, t_2, ... , t_n on an initial state s_0. We see that the output state of each transaction t_i is the input state of the next transaction t_{i+1}. Additionally, we see that the stream of transactions functions produces a stream of observable results r_1, r_2, ... , r_n.

For example, using the transaction functions `get_a`, `get_b` and `swap` that we have just seen, we can execute the sequence of transactions `[get_a, get_b, swap, get_b, ...]` on the state initial state `(1, 2)` using the `tm` function as follows:

```
tm (1, 2) [get_a, get_b, swap, get_b, ... ] → ["1", "2", "", "2", ... ]
```

Assuming that we want multiple users to use the system, we can assume that each user produces a stream of transactions, and expects a stream of results. To obtain a single stream of transactions for the `tm` function, the streams from the users have to be merged into a single stream of transactions. Trinder merges the streams from multiple users non-deterministically. Additionally, we have to distinguish which result in the output stream belongs to which user. To do this, transactions in the merged stream can be tagged by the identifier of the user. The transaction manager can tag a result using the tag of the transaction it processed, such that we know to which user a result belongs. Because we implement our prototype in an imperative language, our approach differs from that of Trinder, so we will not cover this topic in further detail.

What is interesting about the functional transaction processing model is that it is relatively east to guarantee that the ACID properties hold. Serializability is trivially satisfied, as transactions are executed serially. Recoverability can be satisfied by requiring that all transactions are total functions. A *total* function is a function that always produces a result. This also means that transactions have to handle any error that may occur, such as consistency errors. For example, a consistency property on a state can be guaranteed by wrapping a transaction function `t` by a function of the form:

```
if g(t(s)) then t(s) else s
```

Where `s` is the current state, and `g` validates that the consistency properties hold in the new state `f(s)`. Finally, durability can be guaranteed by journaling transactions before executing them, which we discuss in more detail in Chapter 8.

A limitation of this model in practice is that we have to assume that transactions can not abort due to run-time errors such as running out of memory. Another problem in practice is that transactions may be non-terminating, or take a very long time to terminate. However, these issues are out of the scope of this thesis.

3.3. Executing Functional Transactions Efficiently

Now that we have seen how the functional transaction processing model works, we will now discuss the efficient implementation of this model using graph reduction, as shown in the work of Trinder [38], Nikhil [28] and McNally [26].

Using graph reduction, we can represent a state as a graph that has a single root node. When a transaction is applied to a state graph s, a new state graph s' is constructed. The essential idea to maintain efficiency is that if sub-graphs of the state s are used in the construction of s', we share those sub-graphs between s and s' instead of copying the whole graph. Intuitively we could say that in the construction of the new state s', we only encode how s' differs from s.

To make efficient use of sharing, it is neccesary that states have a tree structure, as this allows sharing of whole branches of the tree that are not modified. If instead the state has a sequential structure, such as a list, we are forced to copy most of the state if a single element is modified.

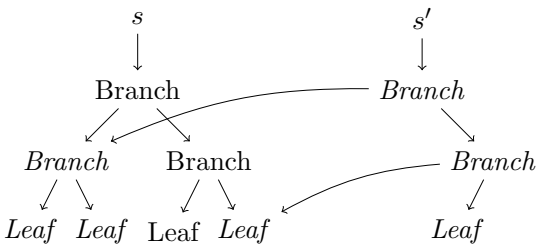


Figure 3.3.: Sharing common sub-graphs between states.

Figure 3.3 shows an example of sharing in action. We see two states, where s and s' are pointers to the root nodes of these states. The initial state s contains some abstract tree data structure. After a transaction updates a single tree element of the state s, we obtain the state s'. We see that only the path from the root of the tree to the updated element is stored in the new state s', and we refer to the old state s for branches of the tree that have not been modified. Nodes reachable from s' are emphasised, showing that the amount of memory required to store the state s' does not increase compared to the amount of memory required to store the state s.

3.4. Executing Functional Transactions Concurrently

In the previous section we have seen that the functional transaction processing model can be implemented efficiently using graph reduction for transactions that are executed serially. However, for acceptable response times in a transaction processing application, we need to be able to execute transactions concurrently. In this section we show how functional transactions can be evaluated concurrently by reducing states lazily, as shown in the work of Trinder [38].

The essential idea for concurrency in functional transaction processing is that while a state s is being constructed by a transaction t, another transaction t' can already start producing a new state s' based on the parts of s that have already been constructed. This idea essentially shows that concurrent evaluation of transactions is possible, as both transactions t and t' are evaluated at the same time.

Taking this idea further, we can first construct those parts of the state s that are required for the construction of the new state s'. This idea can be implemented by constructing states lazily. That is, we only construct parts of the state s if they are required for the construction of state s'. This means that states do not have to be reduced to normal form at all unless their normal form is required. In a transaction processing system, only the observable results of transactions need to be reduced. This means that parts of states only have to be reduced if they are required for the reduction of a transaction result.

We now illustrate this concept through an example. Assume that we have some initial state s, containing some abstract tree data structure. We have a transaction function t_u that takes s and constructs a new state s' by applying a mapping function *map* to the tree in state s, mapping the function f. We also have a transaction t_r applied to the state s' that reads the state by producing as observable result r a boolean representing if some element is contained in the tree using a function *contains*. For simplicity, we omit the details of the function and data structure implementations.

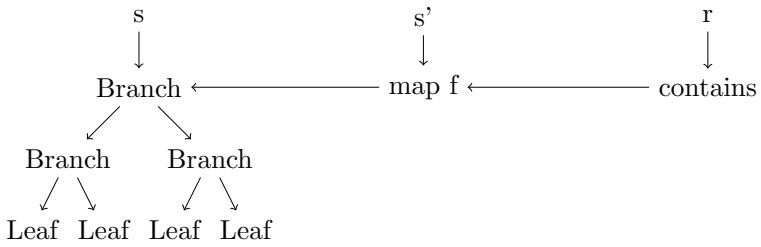


Figure 3.4.: Example system before reduction.

Figure 3.4 shows a graphical representation of the example system before any reduction has taken place. We see that s points to the root node of the initial state, s' points to the root node of the state produced by t_u, and r points to the observable result of t_r.

We see that common parts of s and s' are shared, and we see that common parts of s' and r are shared.

Figure 3.5 shows intermediate steps in the reduction of the observable result r. It can be seen that the parts of s' that are required for the reduction of r are reduced on demand through lazy evaluation. In the first and second reduction step, in order to evaluate *contains* we first need the result of *map* f. In both cases we reduce *map* f to obtain *Branch*, which enables *contains* to be reduced by a single step, selecting one of the branches. In the last reduction step, *map* f evaluates to *Leaf*, allowing the *contains* function to reduce to its final result: *True*.

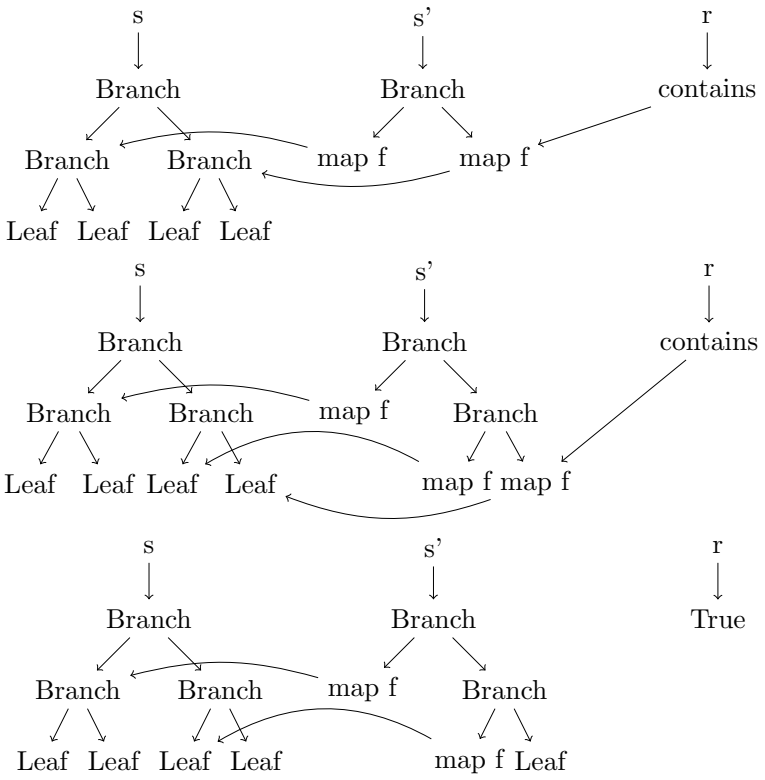


Figure 3.5.: Concurrent evaluation of functional transactions.

The example shows that only those parts of s' are reduced that are actually needed for the reduction of r. More generally, the example shows that a transaction that queries the state may finish before a transaction that updates the complete state has been fully evaluated.

Limitations to Concurrency

The amount of concurrency that can be provided by this method is not unlimited. Trinder [38] was the first to investigate some of the limitations of this form of concurrency, and provided some approaches to overcome these limitations. In this section we will only discuss the limitations to concurrency, and refer the reader to the thesis of Trinder for additional details.

The evaluation of a transaction can be *blocked* by a redex that prevents access to a part of the state while it is being reduced. For example, if we have an expression $E = \mathit{if}\ x\ \mathit{then}\ a\ \mathit{else}\ b$, then either the value a or b is only constructed when the value of x is known. So, all transactions that require access to the result of E are prevented from making progress until the evaluation of x has been completed.

Using the concept of blocking, we can analyse how transactions might affect each other. A transaction that does not modify the state never hinders another transaction, as it does not insert redexes into the state. A transaction that modifies the state without reading from it is itself also never blocked, as the new state is reduced lazily. However, a transaction that modifies the state may hinder transaction that read the state, as the redexes that it inserts into the state may block access to part of the state that another transaction needs to access.

Blocking should present be no problem for redexes whose reduction takes just takes a constant number of steps. An example of such a redex is the application of *map* as in the example in the previous section. However, if the reduction of the redex requires a number of reduction steps dependent on one of its arguments, access to parts of the state could be prevented for a long time.

A concrete example of this problem that Trinder provides is maintaining balance in binary search trees when inserting elements. The problem here is that we only know if we need to re-balance at a certain node after the element has been inserted. This means that for the redex at the root of the tree, we only know what the result is once insertion is completely done, preventing concurrent access to the tree completely during the insertion of an element.

3.5. Transactional Functional Languages

So far we have assumed that the state of the system is of some fixed type `State`, and that a fixed set of transaction functions are implemented by the system. In this section we describe a model for *transactional functional languages*, which provides a more flexible model, allowing the type of the state to change dynamically, and allowing functions to be created dynamically. This model is used in the languages *AGNA* [29] and *STAPLE* [26], and also forms the basis of our prototype language. In this section we first discuss the structure of states, followed by the structure of transactions, and finally we discuss the execution of transactions.

States

A *state* in a transactional functional language is a set of *bindings* from names to expressions, where expressions may refer to other expressions in the state by their name. A state is similar to a regular functional program, except that there is no main function. Transactions allow the state to be changed over time by replacing the state by a new state.

Expressions in the state can represent data as well as functions. A data expressions may include primitive data types such as integers and strings, as well as bulk data types such as lists, trees, relations and graphs. A function expression is a λ-abstraction, which may be used to operate on the data stored in the state. Consider the following example:

```
users  →  Cons "Alice" (Cons "Bob" (Cons "Eve" Nil))
length →  λ list . case list of
              Nil → 0
              (Cons x xs) → 1 + length xs
```

This example shows a state where the expression *users* models a list of user names, and the expression *length* is a function that can be used to compute the length of a list.

Transactions

A *transaction* in a transactional functional language consists of two parts, a *result expression* and an *update environment*.

A *result expression* is an expression to be evaluated in the context of the current state, where the expression may refer to expressions in the state by their name, and which produces the observable result of the transaction. The result expression is in a way similar to the main expression of a functional program, with the main difference being that multiple result expressions may be evaluated over the same state by issuing multiple transactions. Consider the following example:

```
result = length users
```

In this example, the free variables *length* and *users* refer to expressions in the state. When evaluated in the example state as shown in the previous sub-section, this expression reduces to the value 3.

An *update environment* is a set of bindings from identifiers to expressions that replaces the state of system when the transactions is executed. Using update environments, transactions may insert functions into the state that can be used in later transactions. Transactions can also update data expressions in the state by replacing their binding with a new value.

Consider the following example:

$$\begin{array}{rcl} \text{length'} & = & \text{length} \\ \text{users'} & = & \text{Cons "Dave" users} \end{array}$$

In this example, again the free variables *length* and *users* refer to expressions in the state. When evaluated in the state as in the previous sub-section, the *length* function is copied to the new state, and a new username "Dave" is prepended to the list in *users*.

Execution of Transactions

Now that we have seen the structure of transactions and states, we can describe how transactions are executed in this model. Execution consists of two steps, *binding* the transaction to the state, and reducing the result expression. Binding a transaction to the state is performed by resolving names that refer to expression in the state. The bound environment is the state for the next transaction The bound result expression can be reduced to normal form to obtain the observable result of the transaction.

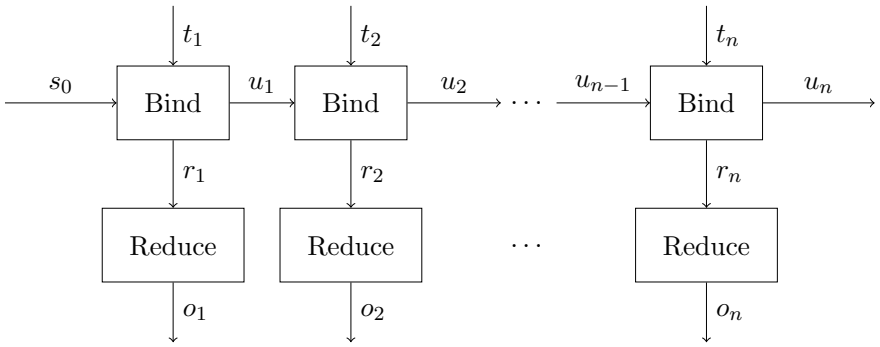


Figure 3.6.: Executing transactions in a transactional functional language.

Figure 3.6 shows the flow of data for the execution of a sequence of transactions t_1, ... , t_n on an initial state s_0. Each transaction t_i is bound to the state, producing a bound update environment u_i, which is the state for the next transaction, and producing a result expression r_i, which is reduced to find the output o_i of the transaction.

3.6. Conclusions

In this chapter we have seen how functional languages can be applied to transaction processing. In particular, we have seen:

- The concept of transactions and transaction processing systems.
- A model for functional transaction processing systems.
- The efficient implementation of functional transaction processing by sharing common parts between states.
- Using lazy reduction of states to allow concurrent execution of functional transactions, and the limitations of this approach.
- A model for transactional functional languages that allows bindings in the state to be created dynamically.

In Chapter 4 we describe our prototype language, which is based on the model for transactional functional languages that we have seen in this chapter. In Chapter 7 we discuss the implementation of a transaction manager that implements the execution of transactions in transactional functional languages. In Chapter 5 we discuss a procedure `resolve` that implements the binding operation to execute transactions, and a procedure `nf` to perform reduction of result expression.

Part II.

Contributions

4. A Prototype Functional Transaction Processing Language

In the previous chapter we have discussed functional transaction processing, and we have seen a model for transactional functional languages. In this chapter we describe a prototype language for functional transaction processing that incorporates these ideas. Our language is similar to a regular functional programming language, but it has additional constructs for transaction processing.

```
<transaction>  ::=  ( <definition> | <stored>)*
     <stored>  ::=  'transaction' <variable> ( '(' <variable>* ')' )?
                    '{' <definition>* '}'
 <definition>  ::=  <variable> ( '(' <variable>* ')' )? '=' <expression>
 <expression>  ::=  LITERAL
                 |  ( <variable> | <constructor> ) ( '(' <expression>* ')' )?
                 |  'match' <expression> '{' ( <pattern> '->' <expression> )+ '}'
                 |  'let' ( <variable> '=' <expression> )* '{' <expression> '}'
    <pattern>  ::=  <constructor> ( '(' (<variable> | '_' )* ')' )?
   <variable>  ::=  'a'..'z' ('a'..'z' | 'A'..'Z')* ''?
<constructor>  ::=  'A'..'Z' ('a'..'z' | 'A'..'Z')*
```

Figure 4.1.: Grammar of prototype language.

Figure 4.1 shows the EBNF syntax of our language, which is explained in the sections below. In the first section, we discuss our expression syntax, which is similar to regular functional languages. Next, we discuss the language constructs for the specification of functional transactions. After that, we discuss stored transactions, which can be invoked through an external interface, and provide a basis for a security mechanism and performance optimisations. Finally, we discuss how our language can be used to construct a transaction processing system with a domain-specific interface.

4.1. Expressions

We start out by discussing the expressions language of our prototype language, which is similar to existing functional programming languages.

Expressions are defined recursively, and can be:

- A *literal*, such as an integer, a floating point number or a string.
- A *function application* of the form f`(`$\texttt{v}_1$ ... $\texttt{v}_n$`)` that denotes the application of the arguments $\texttt{v}_1$ ... $\texttt{v}_n$ to f, where f is either a variable or a data constructor function. If f has no arguments, the parenthesis may be omitted.
- A *match expression* of the form `match` E {P_1 `->` E_1 ... P_n `->` E_n } that denotes a case distinction over the data constructor function obtained by reducing E, and reduces to E_i in P_i `->` E_i if P_i matches the data constructor function. A pattern has the form C`(`v_1 ... v_n`)`, and *matches* a data constructor function of the form D`(`v_1 ... v_m`)` if $C = D$, where m must be equal to n. A pattern of the form C`(`v_1 ... v_n`)` `->` E binds the variables $v_1, \ldots, v_n$ in the expression E. Additionally, each pattern C`(`v_1 ... v_n`)` in a match expression must have a distinct constructor C, which implies that the order of patterns does not affect the semantics of the program.
- A *let binding* of the form `let` v_1 `=` E_1 ... v_n `=` E_n `in` E that denotes that the variables $v_1, \ldots, v_n$ with values $E_1, \ldots, E_n$ are bound in E. A let expression explicitly introduces sharing of the expressions E_1, ..., E_n in E.

Our language supports structured data through *data constructor functions*, as discussed in Section 2.2. A data constructor function is distinguishable from regular functions as data constructor functions start with an uppercase character, where regular function names start with a lowercase character.

Furthermore, our prototype implementation features some built-in functions that may be used in expressions. We have the arithmetic functions `add`, `sub`, `mul` and `div` to operate on primitive numerals. We have a function `equals` that checks for equality of two primitive literals a and b, returning `True` if $a = b$ or `False` if $a \neq b$. We have a function `compare` that compares two primitives a and b and returns `LT` if $a < b$, `EQ` if $a = b$ or `GT` if $a > b$. Finally, we have a special function `seq` that evaluates its first argument to weak head normal form, and returns its second argument. The function `seq` is used to avoid unnecessary laziness in the implementation of certain functions, such as `fold` [30].

4.2. Transactions

Our language differs from regular functional programming languages in that programs are transactions that are executed in the context of a state. In our language, a state consists of a set of bindings of names to closed expressions, as well as a set of stored transactions. A transaction may produce a value as its result by means of a result expression that is evaluated in the context of the state, and a transaction may modify the state by adding, removing or updating bindings and stored transactions.

The system accepts a sequence of transactions as input, where transactions may come from many different users using the system concurrently. Semantically, this sequence of transactions is executed sequentially, thereby ensuring atomicity and isolation. Our implementation of this language enables concurrent execution of transactions, as well as ensuring durability of the effects of executed transactions. The implementation of our language is discussed in the chapters following this chapter.

A *transaction* consists of a set of definitions and a set of stored transactions. A *definition* can be of the form x `=` E where x is variable and E is an expression, or it can be of the form f`(`v_1 ... v_n`)` `=` E to define a function expression $f = \lambda v_1. \ldots \lambda v_n. E$. Stored transactions are discussed in the next section.

In order to support transaction processing, we introduce two kinds of variables: *current state variables* and *next state variables*. To distinguish them syntactically, next state variables are primed. For example, we write `x` to refer to `x` in the current state, and we write `x'` to refer to `x` in the next state. All expressions may refer to both the current state and the next state.

In contrast to the transactional functional language model, a transaction in our language only describes the updates to the state. In order to specify an update to the state, we assign an expression to a next state variable. For example by writing `x' =` E, the value of `x` is E in the next transaction that is executed. We may also assign values to current state variables to define functions that do not go into the state, i.e. that are local to the transaction. E.g. we may write `x =` E, and use the variable `x` in the transaction. We also have a special local variable `result`, which when assigned produces the observable result of the transaction.

Example: A Database of User Names

We now illustrate the use of our language for defining functional transactions by means of some examples. Let us assume that we want to create a database of user names where we want to be able to add user names, see a list of all user names in the database, test if a certain user name is in the database, and see how many user names are in the database.

For simplicity, we represent the set of user names as a list. A more efficient approach is to use an associative map, but this complicates the implementation and strays from our goal of introducing the language. To see a list of user names, we simply fetch the

list. To test if a certain user is in our database, we use a generic function to test if an element belongs to a list. To query how many users there are in our database, we use a generic function to compute the length of a list.

```
1 users' = Nil
2 length'(list) = match list {
3   Nil -> 0
4   Cons(x xs) -> add(1 length'(xs))
5 }
```

Listing 4.1: Setting up the database.

Listing 4.1 shows a transaction that updates the state to include a variable `users`, which is initialised to the empty list, and a function `length` that can be used to compute the length of a list. Note that in the definition of `length'` we refer to `length'` itself to create a recursive function. If we would refer to `length` instead, we would refer to the value of `length` in the current state.

```
1 users' = Cons("bob" users)
2 result = length(users')
```

Listing 4.2: Inserting a user name.

The transaction as shown in Listing 4.2 inserts a user into the database, and requests the size of the resulting database. Note that in the expression `users'` we refer to `users` in the current state; thus inserting a user into the existing database. The observable result of the transaction is the length of `users'`, which includes our newly inserted user "bob" as we refer to `users` in the next state.

```
1 contains(value list) = match list {
2   Nil -> False
3   Cons(x xs) -> match equals(x value) {
4     True -> True
5     False -> contains(value xs)
6   }
7 }
8 result = contains("bob" users)
```

Listing 4.3: Querying the database.

Finally, the transaction shown in Listing 4.3 requests whether a user name is in the database using a locally defined function `contains`. The `contains` function only exists during the execution of the transaction, and is not available to any subsequent transaction.

4.3. Stored Transactions

A stored transaction is a predefined transaction that is stored in the state of the system, and is similar to a stored procedure in traditional database management systems. A stored transaction may be parameterised, and can be invoked multiple times. Stored transactions provide a basis for the construction of a domain specific interface to a functional transaction processing system, as discussed in the next section.

First, the main difference between a stored transaction and a function is that a function defines a transformation over an expression, while a stored procedure defines a transformation over a state. This means that, given a state, a stored transaction can be executed on its own, while a function is part of an expression in a transaction.

The definition of stored transactions is part of a regular transaction, as it modifies the state of the system. A *stored transaction* consists of a name, a list of parameter names, and a *body* that is a transaction. The body of a stored transaction is a regular transaction, but this transaction is only executed when the stored transaction is called. When a stored transaction is *called* with a set of argument bindings, free variables in its body are bound to the current state and the arguments that have been provided.

Consider the following example:

```
1 transaction add_user(name) {
2   result = contains(users name)
3   users' = match result {
4     True -> Cons(name users)
5     False -> users
6   }
7 }
```

This transaction defines a stored transaction called `add_user` that has a parameter `name`. The body of the stored transaction contains a result expression that tests whether `name` is already in the list `users`, and inserts `name` in `users` if the name is not already in the list. This example shows that a stored transaction can enforce a consistency rule, as it enforces that no duplicates are inserted into a list.

A stored transaction can be invoked through an external interface, for example an HTTP interface could be used to invoke the stored transaction from the example as follows:

```
POST /add_user?name=bob
```

The observable result of the stored transaction call can be returned in the body of the HTTP response.

Stored transactions provide a basis for integrating a role based security mechanism into our language. A user can authenticate itself to the system when calling a transaction, and the system can decide based on the role of the user if it is allowed to use that stored transaction.

Additionally, if our language would be extended to support typing, a stored transaction only has to be type checked and pre-compiled when it is inserted into the state. This allows a higher level of performance than using regular transactions, as less work has to be performed per transaction.

4.4. Domain-Specific Interfaces

Being able to invoke stored transactions using an external interface provides a basis for the implementation of a domain-specific interface to an application in our language. Figure 4.2 shows a possible architecture of a system based on this approach. The application consists of a set of stored transactions that implement a domain-specific interface. Additionally, this architecture extends the architecture that was shown in the introduction with a presentation layer. The presentation layer is outside the persistent language, and can be a website, or a physical system such as an ATM.

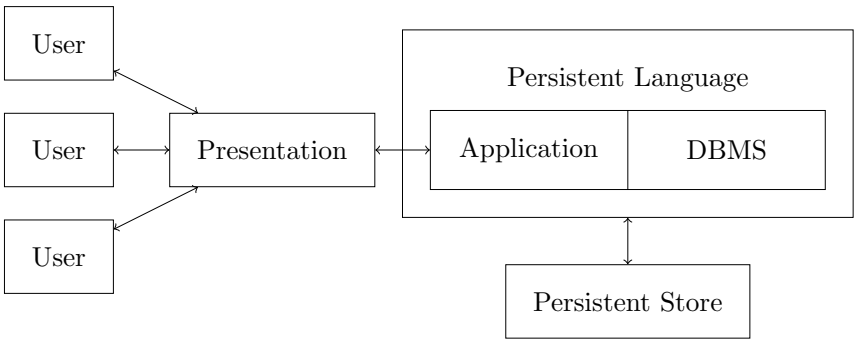


Figure 4.2.: Architecture in our approach.

This system looks similar to the architecture of the traditional approach as discussed in the introduction, where we have a DBMS together with a general purpose programming language. However, the key difference here is that the interface between the presentation layer and the application is domain specific. This means that there is no impedance mismatch, there is no risk of command injection attacks, communication between the presentation layer and the application layer is kept to a minimum, and there are no interactive transactions that may affect concurrency. The only issue remaining is that a system using this architecture is distributed. However, assuming that the correctness of the application has been verified, the only notable failures that can occur in this setting are in the presentation layer, so the integrity of data is not affected.

4.5. Conclusions

In this chapter we have seen a language for the definition of transactions using functional languages. In particular, we have seen:

- The syntax of our language.
- Expressions, data constructor functions and built-in functions in our language.
- Current state variables and next state variables provide a basis for defining functional transactions that operate on a state consisting of a set of bindings.
- Stored transactions can be invoked externally, and provide a basis for security and performance optimisations.
- Stored transactions can provide a basis for the construction of transaction processing systems with a domain-specific interface.

In Chapter 5 we discuss the implementation of a graph reduction that can be used to implement our prototype language. In Chapter 7 we discuss the implementation of a transaction manager for the execution of transactions written in our prototype language.

5. Graph Reduction for Transaction Processing

In this chapter we discuss the implementation of graph reduction for the implementation of functional programming languages. As a basis for our implementation we use template instantiation [32], which we have modified to allow bindings to be created dynamically. We use the graph reducer discussed in this chapter as the basis for a parallel graph reducer, which is discussed in Chapter 6.

In this chapter, we first discuss the notation that we use to describe our implementation. Next, we review template instantiation as a method for the implementation of graph reduction. We then discuss the modifications to template instantiation that are needed to create bindings dynamically. After that, we provide a global overview of our graph reduction implementation, followed by the details of our implementation: resolving free variables, weak head normal form reduction, and finally the reduction of graphs to normal form.

5.1. Preliminaries

Our implementation is written in Java, but for the purpose of conciseness we describe the implementation in a pseudo-language. We now briefly cover the notation that we use in this thesis.

We assume that memory is laid out according to a pointer structure model. A pointer structure consists of a fixed number of named fields that each have a corresponding value. Values can be either pointers to other nodes, primitive data values, or an array of values.

We write `type T = f1:T1 * ... * fn:Tn` to define the type of a node `T` that has fields `f1, ..., fn` of their respective types `T1, ..., Tn`. We write `data T = C1(f1:T1, ..., fn:Tn) | ... | Cn(g1:U1, ..., gn:Un)` to define an enumeration data type `T` with parameterised elements identified by a constructor `C1, ..., Cn`. Given a pointer `x : T` where `T` is an enumeration data type, we can determine if `x` has constructor `C` by writing `x is C`.

In the definition of procedures, we assume that we can pattern match on the constructor of an enumeration data type. For example, if we define a procedure `f(C(x) : T)` { ... } and a procedure `f(x : T)` { ... } where `C` is a constructor of `T`, and we execute

`f(new C(x))`, then `f(C(x) : T)` is invoked. If we execute `f(new D(x))`, then `f(x : T)` is invoked.

We assume that parameters are passed by reference if an argument is a node or array, or passed by value if an argument is of a primitive data type. We write `a` $\leftarrow$ `b` to assign `b` to `a`.

We write `n.f` to access a field `f` of some node `n`. New nodes can be constructed by writing `new T(`v_1`, ..., `v_n`)`, where `T` is the type of the node, and v_1, ..., v_n are values that are assigned to the fields of the node being constructed, in the order of its type definition.

We can construct a new array of type `T` and length `n` by writing `new T[n]`. We write `a[i]` to access the i^{th} element of an array `a`, where the first index of the array is 0. Furthermore we assume that that we can request the length of an array `a` by writing `a.length`.

Finally, for this implementation we are not concerned with memory management. We assume that an unlimited amount of memory is available. In our actual implementation in Java, the garbage collector reclaims memory that is no longer accessible to provide the illusion of having an unlimited amount of memory available.

5.2. Template Instantiation

We now discuss *template instantiation* [32] as a method for the implementation of graph reduction to implement lazy functional programming languages.

The basic idea of template instantiation is that for every function in a functional program we construct a *template graph*. All template graphs are stored globally in a *template map* that maps the name of a function to its template graph. We also have a *reduction graph* where graph reduction is performed. Free variables in both the template graphs and the reduction graph refer to templates in the template map. If a function is to be applied to a sequence of arguments, we look up the template graph for the function, and *instantiate* the template by copying the template while substituting bound variables by the function arguments.

Template instantiation requires that all expressions are supercombinators. A *supercombinator* of arity n is an expression of the form $\lambda x_1.\lambda x_2.\ldots\lambda x_n.E$ where E is not a lambda abstraction, E does not contain free variables except free variables that refer to supercombinators, and any lambda abstraction in E is a supercombinator. Intuitively, a supercombinator does not contain locally defined functions that contain variables that are free in this locally defined function. For example, $\lambda x.(x + ((\lambda y\ .\ x + y)\ x))$ is not a supercombinator, as y is a free variable in the innermost lambda-abstraction.

If we would allow non-supercombinator expressions in the template instantiation approach, a template may contain another template in its body, i.e. a local function. If the outermost template is instantiated, the innermost template will only be partly

instantiated, leaving bound variables in the reduction graph. If we only allow supercombinators, we do not have this problem.

All expressions can be transformed into supercombinators through a process called *lambda lifting* [33]. The basic idea of lambda-lifting is that non-supercombinator lambda-abstractions are 'lifted' to the supercombinator level by introducing additional bindings for their free variables. E.g. in $\lambda x.(x + \lambda y.(x + y))$, the expression $\lambda y.(x + y)$ can be lambda-lifted to obtain the supercombinator $\lambda y.\lambda x.(x + y)$. The implementation of lambda-lifting is outside the scope of this thesis, and we assume that all expressions are already in supercombinator form. This is syntactically enforced by our prototype language, as the grammar does not support locally defined functions.

A special kind of supercombinator is a *constant applicative form* (CAF) supercombinator, which is a supercombinator that does not have any parameters. The template of a CAF supercombinator must not be copied like a *non-CAF* supercombinator, as they are already in their instantiated form. This means that in our implementation, we have to treat CAF supercombinators differently from supercombinators that do have parameters.

Finally, graph reduction in template instantiation is performed by repeatedly finding the next redex to reduce, reducing the redex, and then overwriting the root of the redex with the reduced result. In order to implement lazy evaluation we use outermost reduction, which corresponds to reducing towards weak head normal form. We discuss the implementation of graph reduction in more detail in the remainder of this chapter.

5.3. Adaptations for Dynamic Bindings

Before we discuss the implementation of our graph reducer, we discuss some modifications to standard template instantiation to allow bindings to be created dynamically.

As discussed in the previous section, in regular template instantiation we maintain a template map. If we want to add bindings dynamically, we have to add new templates to the template map. Additionally, bindings may become obsolete and may have to be removed from the template map to reclaim memory. One problem here is that we have to be make sure that every supercombinator has a unique name in the template map. Another problem is that we need a way to find out if a template is no longer used, so that the system can reclaim its memory.

Instead of maintaining a template map, we statically resolve all free variables by replacing them with a pointer to the corresponding template. In doing this, we can drop the template map altogether. Templates are now anonymous, so we do not have to give them unique names. Also, when templates are not refered to anymore, they can be automatically garbage collected by the runtime. Using this modification, we essentially obtain a single graph containing both the templates as well as the reduction graph. An

additional advantage of resolving variables statically is that we do not have to perform a lookup every time we instantiate a template.

Additionally, in normal template instantiation there is a single root node into the reduction graph. However, for our transaction processing language, we keep track of multiple pointers into the reduction graph, corresponding to the different bindings in the state.

A non-CAF supercombinator template in our implementation starts with a special supercombinator node, as discussed in the next section. During the instantiation of a supercombinator template, if we encounter such a supercombinator node, we know that a non-CAF template starts and the current template ends. Supercombinators that are CAF's do not begin with a supercombinator node, as that would require that they are part of a redex, which is not the case. In order to know that a template ends in such a case, we insert a special *symbol node* that marks the end of a template when we resolve a free variable that refers to a CAF supercombinator.

5.4. Implementation Overview

Now we provide an overview of our implementation. We dicuss the data structures that we use, and we provide a global overview of the procedures that we use for reducing graphs built from these data structures.

Data Structures

```
1  data Node = // Lambda Calculus
2              SuperCombinator(template : Node)
3            | BoundVariable(index : Integer)
4            | FreeVariable(id : String)
5            | Symbol(binding : Node)
6            | Application(function : Node, arguments : Node[])
7            // Sharing
8            | Sharing(shared : Node)
9            // Structured data
10           | Data(constructor : String)
11           | Case(cases : Map String Node)
12             // Primitives
13           | Int(value : Integer)
14           | Double(value : Double)
15           | Add(left : Node, right : Node)
16           | Sub(left : Node, right : Node)
17           | ...
18
19 type Environment = Map String Node
```

Listing 5.1: Data Structures for our graph reducer.

Listing 5.1 shows an overview of the data structures that we use to represent graphs. In the paragraphs below, we discuss these data structures in more detail.

A *supercombinator node* represents a lambda-abstraction of the form $\lambda x_1 . \ldots . \lambda x_n . E$ where the template of the supercombinator node corresponds to the expression E. A template is built from the same nodes as that we use for graph reduction. We do not keep track of the number of arguments in a supercombinator node, as we do not support partial function application. In our implementation, a supercombinator node is essentially a primitive function.

A bound variables is represented by a *bound variable node*. We use *de-bruijn indices* [10] instead of named variables, such that a variable with index n refers to the nth parameter of its enclosing supercombinator. When a supercombinator is instantiated, an array of arguments is passed where the ith element corresponds to the ith argument of the supercombinator. Bound variables only occur in template graphs, as they are resolved during instantiation of a template.

A free variable is represented by a *free variable node*, which has a name that refers to a binding in the state. Free variables only occur during compile-time, as they are resolved before reduction. If a free variable resolves to a CAF supercombinator, it is replaced by a *symbol node* that refers to the appropriate CAF template, thereby marking the boundary of the supercombinator template that the free variable is part of. If a free variable resolves to a non-CAF template, no symbol node is inserted, as the supercombinator root node of the referred template marks the boundary of the enclosing template.

An application in the λ-calculus is represented by an *application node*, which consist of a primitive function and an array of argument nodes. An application node marks the start of a redex, where the second part of the redex consists of the primitive function of the application node, such as a supercombinator node, a case node or a primitive arithmetic function.

One of the key features of graph rewriting is that we can overwrite the root of a redex with its result. However, Java does not allow us to physically overwrite a node, i.e. replace a node instance by another node instance. To solve this, we use a special *sharing node* that has a pointer that either points to a redex, or the weak head normal form of that redex. Reducing a sharing node to weak head normal form consists of reducing the shared node to weak head normal form, and then updating the pointer of the sharing node to point to the result. If the node is visited a second time, the already reduced result is used instead of reducing the result again. In our implementation we insert a sharing node before every application node, i.e. before every redex.

Structured data is implemented through data nodes and case nodes. A *data node* represents a data constructor function, and holds the name of the constructor together with an array of child nodes. A *case node* is a primitive function that performs a case distinction on the constructor of a data node by selecting one of the alternative cases based on the constructor of a data node applied to it.

Primitive data types are implemented as nodes that *box* a primitive value. Our implementation supports integers, doubles and strings. Primitive operations on primitive data types are implemented as primitive function nodes, including arithmetic functions and comparison functions.

Finally, we have a data type *Environment* that maps names to templates in the reduction graph, to keep track of bindings. As discussed in section 5.3, if the binding is a non-CAF supercombinator, then the root node of the template is a supercombinator node, and if the binding is a CAF supercombinator, then the root node of the template is not a supercombinator node.

Procedures

Now that we have seen the data structures that we use to represent graphs, we can discuss the procedures to implement graph reduction. Our implementation consists of the following procedures:

`resolve` **: Environment** $\rightarrow$ **Environment** $\rightarrow$ **Environment**
: Resolves the free variables in an environment to obtain an environment where all pointers to free variables in all expressions have been replaced by pointers to their respective template. This procedure also accepts an environment of additional bindings that can be bound to the first environment.

`whnf` **: Node** $\rightarrow$ **Node**
: Reduces a node to weak head normal form, thereby implementing lazy graph reduction.

`apply` **: Node** $\times$ **Node[]** $\rightarrow$ **Node**
: Invokes a primitive function given an array of arguments.

`instantiate` **: Node** $\times$ **Node[]** $\rightarrow$ **Node**
: Instantiates a given node that is part of a supercombinator template, the array of arguments corresponds to the parameters with which the supercombinator is invoked, and which are to substitute the bound variables in the template.

`nf` **: Node** $\rightarrow$ **Node**
: Reduces a graph to normal form by repeatedly finding the next redex in the input graph and reducing it to weak head normal form.

In the following sections, we discuss the implementation of these procedures in the order as shown here.

5.5. Resolving Free Variables

First, we discuss our algorithm for resolving free variables in an environment of supercombinators, which is implemented in the `resolve` procedure.

A pointer to a free variables is resolved by replacing it with a pointer to the resolved version of the supercombinator corresponding to the free variable. We want to resolve

templates non-destructively, as to keep the template that is being resolved intact, which is required to re-use templates in stored transactions.

The basic idea of our algorithm is that we copy the original template, while replacing pointers to free variables by pointers to their corresponding resolved supercombinator nodes.

```
1 resolve(FreeVariable(id) : Node, bindings : Environment) : Node {
2   binding ← bindings.get(id);
3   if(binding = null) {
4     return this;
5   } else if(binding : Node.SuperCombinator) {
6     return binding;
7   } else {
8     return new Symbol(binding);
9   }
10 }
```

Listing 5.2: Resolving free variables.

Listing 5.2 shows the implementation of `resolve` for free variable nodes. The `binding` environment contains the resolved versions of the supercombinators that free variables may refer to. We look up the binding for the free variable node in the provided environment. If there is no binding, we can not resolve yet, and we return the free variable node itself. If there is a binding that is a supercombinator node, then we have a non-CAF node, and we return that node. Otherwise the binding is a CAF supercombinator, and we create a symbol node that refers to the binding.

```
1 resolve(Application(function, arguments),
2         bindings : Environment) : Node {
3   resolved_arguments ← new Node[arguments.length];
4   for(i ← 0; i < arguments.length; i ← i + 1) {
5     resolved_arguments[i] ← resolve(arguments[i]);
6   }
7   return new Application(
8     resolve(function),
9     resolved_arguments
10  );
11 }
```

Listing 5.3: Resolving application variables nodes.

Listing 5.3 shows the implementation of `resolve` for application nodes. We resolve the function and arguments of the application node, and we use these to build a new application node as the result. For other nodes resolving is done similar to the application node. That is by constructing a copy of the node where its child nodes have been resolved.

One complication in the use of this algorithm is the construction of the `binding` environment. A supercombinator body may recursively refer to itself, and we can not resolve a supercombinator without having a resolved version of itself available. Our solution to this problem is to copy the root nodes of the supercombinators, and construct an environment of these copies. We then resolve all templates using this environment of copied root nodes. After resolving we update the copied root nodes in-place to match the resolved version.

```
1 resolve(bindings : Environment, additional : Environment) : Node {
2   result ← new Environment<String, Node>();
3   // Copy roots
4   for((name, node) ∈ bindings) {
5     result.put(name, copy(node));
6   }
7   // Resolve free-variable-only supercombinators
8   for((name, node) ∈ bindings) {
9     seen ← {};
10    while(node is FreeVariable) {
11      if(node.id ∈ seen) {
12        throw new Exception("Cyclic reference");
13      }
14      seen ← seen ∪ { node.id };
15      if(result.contains(node.id)) {
16        node ← result.get(node.id);
17      } else {
18        node ← additional.get(node.id);
19      }
20    }
21    for(id ∈ seen) {
22      result.put(id, node);
23    }
24  }
25  // Update copied roots by their resolved variant
26  for(name ∈ bindings.keySet()) {
27    node ← result.get(name);
28    update(node, node.resolve(result ∪ additional));
29  }
30  return result;
31 }
```

Listing 5.4: Resolving free variables in an environment.

There is one special case we have to handle in this approach: a CAF supercombinator that has a free variable as its root node. A free variable gets replaced by a pointer to a supercombinator, and we can not update a copy of it to reflect this change. To handle this

case, we resolve such free variables before resolving any other supercombinators.

An additional complication is that such a *free-variable-only supercombinator* (FVO) may refer to another FVO. This implies that there may be a cycle of FVO's. We can handle this by following the path of FVO's until we find a supercombinator that does not have a free variable as its root node. We can then resolve all these FVO's by pointing them to the the non-FVO node. When implementing this, we have to be careful, as there might be a cycle of FVO's. To detect such cycles, we can remember the FVO's that we have already seen, and terminate when we detect that we have visited an FVO twice.

Listing 5.4 shows the algorithm for resolving environments, incorporating the solutions we have discussed above. The `copy` procedure only copies the root node of the expression, i.e. a shallow copy is performed. The `update` procedure updates a node by copying values of the fields from the node passed to it. As an example, Listing 5.5 shows how an application node is updated. This algorithm also allows an environment of additional bindings `additional` to be bound, which is needed for transaction processing to bind free variables in a transaction to state variables.

```
1 update(Application(dst_fun, dst_arg) : Node,
2        Application(src_fun, src_arg : Node) : Void {
3   dst_fun ← src_fun;
4   dst_arg ← src_arg;
5 }
```

Listing 5.5: Updating a node.

The result of the execution of this algorithm is that we have an environment of supercombinators that do not contain free variables. Essentially, all entries of this environment map are roots into a single graph.

5.6. Weak Head Normal Form Reduction

We now discuss the reduction of redexes to weak head normal form (WHNF). As discussed earlier, WHNF reduction corresponds to lazy evaluation of functional expressions. We use WHNF reduction in order to implement normal form reduction, as is discussed in the next section.

The implementation consists of three parts: the `whnf` procedure, which is called to reduce a node to weak head normal form, the `apply` procedure, which is called by the `whnf` procedure when applied to a application node to invoke a primitive function, and finally the `instantiate` procedure, which is called by the `whnf` procedure when applied to a supercombinator node to instantiate a template. We discuss the implementation of these procedures in sequence. In these algorithms we assume that the executed program is well-typed, so that we do not have to handle type errors during execution.

```
1 whnf(Application(function, arguments) : Node) : Node {
2   return whnf(apply(whnf(function), arguments));
3 }
4
5 whnf(Sharing(shared) : Node) : Node {
6   shared ← whnf(shared);
7   return shared;
8 }
9
10 whnf(node : Node) : Node {
11   return node;
12 }
```

Listing 5.6: The `whnf` procedure.

Listing 5.6 shows the implementation of the `whnf` procedure. There are only three cases to be handled. If we encounter an application node, then we have found a redex; we reduce the function component of the application node, invoke the `apply` procedure to execute the primitive function, and reduce the result of `apply` to weak head normal form. The arguments are not reduced in the application node, as they are evaluated lazily. If we encounter a sharing node, we reduce the shared node, and store the reduction result so that the next time we encounter this node we do not have to reduce the shared node again. All other node types are already in weak head normal form, so we can just return the node itself as the result.

```
1 apply(SuperCombinator(template) : Node, arguments : Node[]) : Node {
2   return instantiate(template, arguments);
3 }
4
5 apply(Symbol(binding) : Node, arguments : Node[]) : Node {
6   return apply(binding, arguments);
7 }
8
9 apply(Sharing(shared) : Node, arguments : Node[]) : Node {
10   return apply(shared, arguments);
11 }
12
13 apply(Case(cases) : Node, arguments : Node[]) : Node {
14   return cases(whnf(arguments[0]).constructor);
15 }
16
17 apply(application : Node.Application,  arguments : Node[]) : Node {
18     return apply(whnf(application), arguments);
19 }
```

```
20 apply(Add : Node, arguments : Node[]) : Node {
21   return new Int(
22     whnf(arguments[0]).value +
23     whnf(arguments[1]).value
24   );
25 }
```

Listing 5.7: The `apply` procedure.

Listing 5.7 shows the implementation of the `apply` procedure, which is invoked by the application node. For a supercombinator node, we instantiate its template using the `instantiate` procedure, passing the arguments as template parameters. For a symbol node, we propagate the application to the bound node. Similarly, for a sharing node we propagate the application to the shared node, we can assume that the shared node has already been reduced because apply is only called from an application node, which first reduces the function node before calling it. For a case node, we eagerly evaluate the first argument to obtain a data node, and return the corresponding case from the `cases` mapping. If we encounter an application node then we have found a redex that must be reduced first before reducing the original application node, so we call `whnf` on this redex, and then propagate `apply` to the reduced result of that redex. Finally, for a primitive addition node, we reduce both operands to obtain two integer nodes, we unbox the integers, then we perform the addition natively and box the result to obtain an integer node as the final result.

Finally, listing 5.8 shows the implementation of the `instantiate` procedure for some nodes. The essential idea is that we copy the template, substituting bound variables by their corresponding argument. Instantiation is terminated when we encounter the boundary of the supercombinator, which is the case if we encounter a supercombinator node or a symbol node.

```
 1 instantiate(sc : Node.SuperCombinator, arguments : Node[]) : Node {
 2   return sc;
 3 }
 4
 5 instantiate(Symbol(binding) : Node, arguments : Node[]) : Node {
 6   return binding;
 7 }
 8
 9 instantiate(BoundVariable(index) : Node, arguments : Node[]) : Node {
10   return arguments[index];
11 }
```

```
12 instantiate(Sharing(shared) : Node) : Node {
13   return new Sharing(instantiate(shared));
14 }
15
16 instantiate(Application(function, app_args) : Node,
17              arguments : Node[]) : Node {
18   ia ← new Node[arguments.length];
19   for(i ← 0; i < arguments.length; i ← i + 1) {
20     ia[i] ← instantiate(app_args[i], arguments);
21   }
22   return new Application(
23     instantiate(function, arguments),
24     ia
25   );
26 }
```

Listing 5.8: The `instantiate` procedure.

5.7. Normal Form Reduction

In the previous section we discussed reduction to weak head normal form. We now use this to implement normal form reduction. Normal form reduction is used in our prototype to reduce the result expressions of transactions. We also use normal form reduction for reducing states to normal form, so that we can create a consistent snapshot of the state as is discussed in Chapter 8.

```
1 nf(root : Node) : Node {
2   nf_root ← whnf(root);
3   if(nf_root is Data) {
4     // Construct an array to store the reduced children
5     reduced_children ← new Node[nf_root.children.length];
6     // Reduce children to normal form
7     for(i ← 0; i < nf_root.children.length; i ← i + 1) {
8       reduced_children[i] ← nf(nf_root.children[i]);
9     }
10    // Construct a new data node
11    nf_root ← new DataNode(
12      nf_root.constructor,
13      reduced_children
14    );
15  }
16  return nf_root;
17 }
```

Listing 5.9: Normal form reduction algorithm.

Listing 5.9 shows the implementation of the `nf` procedure for normal form reduction of graphs. Given the root of the graph to be reduced, we first reduce the root to WHNF using `whnf`. There is actually only one case where `whnf` does not return a result in normal form, that is when it returns a data node. To reduce such a data node to normal form, we recursively reduce its children to normal form using `nf`, and construct a new data node as the result.

5.8. Conclusions

In this chapter we have seen the implementation of a graph reducer that has been modified for transaction processing. In particular, we have seen:

- Template instantiation as a method for implementing graph reduction.
- An adaptation to standard template instantiation to allow bindings to be created and removed dynamically.
- Data structures for our implementation of graph reduction.
- An algorithm for resolving free variables statically.
- Algorithms for lazy graph reduction.
- An algorithm for reduction of graphs to normal forms.

The graph reducer as discussed in this chapter provides a good basis for the implementation of a transactional functional language. Using this graph reducer, we can already execute updates to the state concurrently due to lazy evaluation of states. In Chapter 6 we adapt the graph reducer to correctly handle concurrent graph reduction, as well as proving a load balancing mechanism to enable parallel graph reduction. In Chapter 7 we use the procedures as described in this chapter to implement a transaction manager for a transactional functional language.

6. Parallel Graph Reduction by Randomisation and Sharing Results

The graph reducer as discussed in the previous chapter already provides a good basis for the implementation of a transactional functional language, and it even allows concurrent execution of update transactions by reducing states lazily. If we want to execute read transactions concurrently, multiple threads may reduce the same state graph concurrently. We need to adapt our serial graph reducer to allow concurrent reduction, by ensuring that results are correctly shared between threads.

We can also develop this method further by incorporating load balancing to enable parallel graph reduction. Load balancing tasks among available hardware resources is the main challenge in parallel graph reduction, because tasks are very fine grained and we do not want to consume too many resources to do load balancing. Our method of load balancing is based on randomising the reduction order of reduction threads. This method is different from methods found in the literature, which are commonly based on a work-stealing approach [35].

In this chapter, we first discuss some preliminaries about concurrent programming. We then describe the idea of randomisation and result sharing in an abstract setting. Next, we apply this abstract model to graph reduction, and provide a short overview of the implementation. We then describe the implementation of result sharing and randomisation for weak head normal reduction. Finally, we describe the implementation of result sharing and randomisation for normal form reduction.

6.1. Preliminaries

To start out, we discuss some preliminaries about concurrent programming. First we discuss concurrency and parallelism, then we discuss threads as a method to introduce concurrency into programs, and then we discuss concurrent operations on shared memory.

To distinguish between concurrency and parallelism, we follow the definitions by Ben-Ari [6]. We say that tasks can be performed *concurrently* if multiple tasks can make progress at the same time. Tasks are performed in *parallel* if multiple tasks are actually making progress at the same time. The purpose of concurrency is to ensure that tasks can complete without depending on the completion of another task. The purpose of parallelism is to execute a set of tasks faster by executing multiple tasks at the same

time using parallel hardware. In the context of transaction processing, our primary aim is to execute transactions concurrency, as we want to be able to execute a transaction without depending on the completion of previously executed transactions. However, being able to execute transactions in parallel is also beneficial, as it improves the overall performance of the system.

A sequential program can introduce concurrency through *threads*. Each thread maintains its own instruction pointer and call stack, and may be executed independently of other threads. Multiple threads can be executed concurrently by either interleaving their execution, or by executing them in parallel using parallel hardware. Which of these methods is chosen depends on the operating system and the hardware on which the program runs.

All threads of a program work concurrently on the same shared memory. In order to reason about concurrent access to shared memory, we assume that all operations on shared memory are atomic. We also assume the existence of an atomic *compare-and-set* operation `compareAndSet(l, c, s)` that atomically executes:

$$\texttt{if l = c then l} \leftarrow \texttt{s}.$$

Additionally, modern hardware may perform instruction reordering to optimise the performance of a program, however the algorithms presented in this thesis assume that there is no instruction reordering.

6.2. Parallelism in Functional Languages

As we have discussed in Chapter 2, functional programs are inherently parallel. However, in practice it is hard to take advantage of the implicit parallism in functional programs. In this section we provide a short overview of the problems in parallelising functional programs.

One problem is that reduction tasks have to be load balanced among the available processors. A common approach to load balancing in parallel graph reduction is *work stealing* [35]. In this approach, every execution thread maintains a work pool. If an execution thread does not have any more work to perform, it may 'steal' a task from the work pool of another execution thread. A problem in this approach is that a functional program may contain such a high amount of parallelism that load balancing produces a large amount of overhead, which may slow the system down instead of speeding it up [20]. A common solution to solve this problem is to limit the amount of parallelism in a functional program. This can be done by annotate parts of the program to introduce parallelism explicitly instead of using the inherit parallelism of a functional program. However, a problem in this approach is that it is hard to determine which parts to annotate for optimal performance.

Another problem is that lazy evaluation forces a parallel functional program to be sequentially executed [37]. Again, a solution to this problem is to annotate programs

to explicitly, however now the goal is to increase parallelism instead of reducing parallelism [24].

In the remainder of this chapter we discuss our load balancing method. Our approach focuses on the issue of load balancing, and not on the problems caused by laziness.

6.3. Randomisation and Result Sharing

In this section we discuss randomisation and result sharing as a method for load balancing in an abstract setting.

Assume that we have some tree where nodes represent deterministic tasks which execution produces as a result a value. Each tasks depends on the results of the tasks below it in the tree, and the main goal is to execute the task at the root of the tree. In order to execute all tasks, a thread can make a depth first walk along all tasks in the tree, and perform a task as it is about to move up from a node to its parent node. I.e. the order of executing the tasks corresponds to a postorder depth-first traversal of the tree.

We can parallelise the work among multiple threads by making each thread take a different walk through the graph, i.e. randomising their execution paths. Once a thread has computed the result of a task, it stores the result of the task so that other threads can see the result. If a thread comes across a task for which a result has already been computed by another thread, it uses that result instead of computing the result itself. That is, threads share results between each another.

Using this method, multiple threads may be working on the same task at the same time. As threads only communicate through result sharing, we have to assume that executing one tasks concurrently by multiple threads presents no problems. Each thread produces its own result, which are all identical because tasks are deterministic. In the end only one of these result is needed, so when we have multiple results, all but one of them is discarded. This means that task execution is a speculative operation, as there might be other threads that are executing the same task at the same time, making the work of all but one of them redundant. This means that if the threads are not balanced well over the tree, there may be a high amount of overhead due to redundant task execution. This also implies that this method may not work well for trees that are very skewed or have only one branch at every node.

The idea of randomisation and result sharing for load balancing of work in parallel systems is not new, and has already been applied successfully in the context of model checking for the parallel exploration of a state-space [16], however to our knowledge this method has not yet been applied to graph reduction.

6.4. Randomisation and Result Sharing for Graph Reduction

Now we discuss how randomisation and result sharing can be applied to graph reduction, and how this can be implemented. As a starting point for the implementation we use

the graph reducer as discussed in Chapter 5.

We start out by mapping concepts in graph reduction to concepts in the abstract model for randomisation and result sharing as described in the previous section. First, we observe that redexes correspond to tasks, and the goal of graph reduction is to find the normal form or weak head normal form of the root redex of the graph. The abstract model assumes that concurrent execution of one task by multiple threads presents no problems. For our serial graph reducer this is indeed the case, as the graph being reduced is immutable because each the reduction of a redex produces a result graph without modifying the existing graph (with the exception of result sharing, as we discuss in a moment).

The walk performed by a reduction thread under lazy evaluation depends on the order in which it reduces redexes on which the reduction of another redex depends. A redex depends on another redex if it requires the result of the other redex in order to be reduced. This is the case for functions that are strict in their arguments, i.e. functions that require evaluation of their arguments. An example of a strict function is primitive addition, if we want to reduce $a + b$, we first have to reduce a and b beforce being able to perform the addition itself. This means that randomisation of walks through the tree can be performed by randomising the choice of which argument to reduce first when applying strict functions. We discuss how this can be implemented in Section 6.6.

The concept of sharing results is already a part of graph reduction, as redexes are replaced by their results in the graph reduction model. In our current graph reducer, sharing nodes are used for this purpose. However, our sharing node implementation so far is not suitable for concurrent reduction. In the next section we discuss why this is, and we provide a solution.

6.5. Result Sharing in Weak Head Normal Form Reduction

In this section we discuss the sharing of results between reduction threads. As a starting point for sharing results between threads, we take the result sharing nodes that we have introduced in our serial graph reducer as discussed in Chapter 5. We modify the `whnf` procedure for this node to allow correct sharing between threads.

```
1 whnf(Sharing(shared) : Node) : Node {
2   shared ← whnf(shared);
3   return shared;
4 }
```

Listing 6.1: Serial graph reduction result sharing.

Consider the procedure for reducing a sharing node to weak head normal in Listing 6.1, as discussed in Chapter 5. If this procedure is executed by one thread, and later another thread executes it, the second thread uses the result produced by the first thread, thereby effectively sharing results between threads. However, if two threads where to invoke this procedure concurrently on the same sharing node, what may happen then?

Let us look at a sample execution. Thread t_1 computes `whnf(node.shared)` to obtain some result r_1, and another thread t_2 concurrently computes `whnf(node.shared)` and obtains the result r_2. Now t_1 updates the sharing node and returns r_1 as the result, and t_2 updates the sharing node and returns r_2 as the result. So, there are now two results r_1 and r_2 which are equivalent, but which are not shared.

While the result of each thread is correct, this example demonstrates that a loss of sharing occurs when two threads reduce a redex simultaneously. This loss of sharing has two effects on the execution of a program. One effect is that memory is wasted due to two equivalent results being used by two threads. Another more serious problem occurs when sharing is lost on a redex. First, computational time is wasted as two threads have to reduce this redex separately, and second the result produced by these two threads, while equivalent, will also not be shared. This may lead to a large computation being executed by two threads separately.

In order to maintain sharing, we have to ensure that each thread that calls the `whnf` procedure on a sharing node produces not only the same result, but we also have to ensure that the result is shared between the threads. The following algorithm ensures this (however, it is still not completely correct, as we discuss later):

```
1 whnf(Sharing(shared) : Node) : Node {
2   local ← shared;
3   reduced ← whnf(local);
4   if(local ≠ reduced) {
5     compareAndSet(shared, local, reduced);
6     return shared;
7   } else {
8     return local;
9   }
10 }
```

Listing 6.2: Maintaining sharing in `whnf` for sharing nodes.

This procedure first fetches the shared node from the sharing node and stores it in `local` so that it is thread local. It then reduces `local`, to obtain its weak head normal form, which is stored in `reduced`. Now, if `reduced` = `local`, then `local` is already in weak head normal form, and we can return it as the result of the procedure. If `reduced` ≠ `local` we have actually performed a reduction, and we use `compareAndSet` to atomically set `node.shared` to `reduced` only if `node.shared` = `local`. If this operation succeeds then `node.shared` contains `reduced`. If the compare and set fails, then another thread must have updated `node.shared` with its result, so `node.shared` contains the result we want to use, as to maintain sharing. The uniqueness of the result is thus guaranteed by this algorithm.

As already mentioned, there is still a problem with this algorithm, which appears in conjunction with the `whnf` algorithm for application nodes. Consider the algorithm for

weak head normal reduction of application nodes as discussed in Chapter 5:

```
1 whnf(Application(function, arguments) : Node) : Node {
2   return whnf(apply(whnf(function), arguments));
3 }
```

Listing 6.3: Implementation of `whnf` in serial graph reduction.

The expression `apply(whnf(node.function), node.arguments)` may create a new redex, for example if `whnf(node.function)` returns a supercombinator node. If two threads invoke the `whnf` procedure on an application node concurrently, two redexes are actually created. In order to maintain sharing, these redexes must be shared, but in our current algorithm they are not. Sharing of the final result will still be maintained by a parent sharing node, so the final result will be shared, but we still have the issue of duplicate computation in the intermediate computation.

The solution to this problem is to not perform the weak head normal form reduction of the redex in the application node, but to do this in the sharing node, as to share intermediate results. For this, we introduce an additional procedure `reduce` that performs a minimal reduction step towards weak head normal form. The implementation of `reduce` is as follows:

```
1 reduce(Application(function, arguments) : Application) : Node {
2   return apply(whnf(function), arguments);
3 }
4
5 reduce(node : Node) : Node {
6   return whnf(node);
7 }
```

Listing 6.4: Implementation of the `reduce` procedure.

The only difference from the `whnf` procedure for application nodes is that we do not perform a tail-recursive call to `whnf`. If `reduce` creates a redex, it is passed to the procedure that invoked `reduce`. In general, if the `whnf` procedure performs a tail-recursive call to `whnf`, then the `reduce` procedure can be constructed by taking the `whnf` procedure and ommitting this tail-recursive call. For all other nodes, `reduce` can simply return the weak head normal form of the node using the `whnf` procedure.

Using the `reduce` procedure, we can now update the `whnf` procedure for sharing nodes, to share intermediate results between threads, as shown in Listing 6.5. This algorithm differs from the previous in that we now perform multiple reduction steps to obtain a weak head normal form, while sharing intermediate results with other threads. If the compare and set operation succeeds then `node.shared` contains `reduced` and we can continue reduction with that value. If the compare and set fails, then another thread must have updated `node.shared` with a reduced form of `local`, and to maintain sharing we continue reduction with that value instead.

```
1 whnf(Sharing(shared) : Sharing) : Node {
2   local ← shared;
3   reduced ← reduce(local);
4   while(local ≠ reduced) {
5     if(compareAndSet(shared, local, reduced)) {
6       local ← reduced;
7     } else {
8       local ← shared;
9     }
10    reduced ← reduce(local);
11  }
12  return local;
```

Listing 6.5: Maintaining intermediate result sharing.

Again, this procedure guarantees the uniqueness of the final result. Also, this algorithm is guaranteed to terminate if reduction of the shared node terminates. There is always one thread that succeeds in reducing the shared node by one step and updating `node.shared`. Any other threads will fail to update `node.shared` with their result using compare and set, and they will continue using the result from the thread that succeeded. Therefor, there is always at least one thread that makes progress towards reducing the shared node.

Using the result sharing algorithms as discussed in this section, we can reduce a graph concurrently using multiple threads, and be ensured that if multiple threads reduce the same redex, they obtain a shared result.

6.6. Randomisation in Weak Head Normal Form Reduction

We now discuss randomisation of the execution paths of threads so that we can distribute work among available hardware threads, enabling parallel graph reduction.

In our current graph reducer, the points where a thread has a choice in the order of reduction of redexes is in the `apply` function for primitive functions that are strict. Consider for example the implementation of `apply` for primitive addition:

```
1 apply(Add : Node, arguments : Node[]) : Node {
2   return new Int(
3     whnf(arguments[0]).value +
4     whnf(arguments[1]).value
5   );
6 }
```

Listing 6.6: Implementation of the addition node.

In our current implementation, each thread first reduces `argument[0]`, followed by `argument[1]`. In order to randomise this, threads must somehow make different decisions on which argument to reduce first. With two arguments we only have two possible orderings, so we can make a decision based on a boolean value whether to reduce from left to right, or from right to left. This gives us the following algorithm, where we have to fill in the dots to make a decision about whether to reduce from left to right, or from right to left:

```
1 apply(Add : Node, arguments : Node[]) : Node {
2   left_to_right = ...;
3   if(left_to_right) {
4     l ← whnf(arguments[0]);
5     r ← whnf(arguments[1]);
6   } else {
7     r ← whnf(arguments[1]);
8     l ← whnf(arguments[0]);
9   }
10
11  return new Int(l.value + r.value);
12 }
```

Listing 6.7: Ordering in the addition node.

There are different strategies we could use to make a decision about reduction order in the example above. One strategy is the use of a thread local psuedo-random number generator to generate random values for each thread. However, we could also attempt to distribute threads somewhat evenly among the left and right branch by alternating the reduction order of threads visiting a node.

```
1 data Node = ...
2           | Add(left_to_right : Boolean)
3           | ...
4
5 apply(Add(left_to_right) : Node, arguments : Node[]) : Node {
6   if(negateAndGet(left_to_right)) {
7     l ← whnf(arguments[0]);
8     r ← whnf(arguments[1]);
9   } else {
10    r ← whnf(arguments[1]);
11    l ← whnf(arguments[0]);
12  }
13
14  return new Int(l.value + r.value);
15 }
```

Listing 6.8: Randomisation in the addition node.

For our prototype implementation, we have chosen to randomise the order of threads by attempting to distribute them evenly among branches. This is implemented by adding a boolean flag to a primitive function node, and every time this node is visited, the flag is negated. The algorithm in Listing 6.8 shows the implementation of this for the primitive addition function. We assume the existence of a procedure `negateAndGet` that atomically negates and gets the passed argument.

In order for this method to work well in practice, we have to ensure that each application node has its own Add node instead of using a singleton Add node. If we would use a singleton node, there will be a lot of communication overhead between hardware due to contention for a single cache line.

Additionally, we could perform the negation of `left_to_right` non-atomically, as in practice this is quite an expensive operation. By doing this, there is a data race [27] when negating the `left_to_right` node, as it is not an atomic operation. In practice this means that multiple threads may choose the same branch, where they would otherwise distribute evenly among branches. Correctness of the result and sharing are not affected when doing this, only the distribution of threads in the graph is affected. In practice there is a high probability that there is a good distribution of threads. As the goal of parallel graph reduction is to maximise performance, using a slow atomic operation here may not be the best option.

```
1 nf(root : Node) : Node {
2   nf_root ← whnf(root);
3   if(nf_root is Data) {
4     // Construct an array to store the reduced children
5     reduced_children ← new Node[nf_root.children.length];
6     // Reduce children to normal form
7     for(i ← 0; i < nf_root.children.length; i ← i + 1) {
8       reduced_children[i] ← nf(nf_root.children[i]);
9     }
10    // Construct a new data node
11    nf_root ← new DataNode(
12      nf_root.constructor,
13      reduced_children
14    );
15  }
16  return nf_root;
17 }
```

Listing 6.9: Normal form reduction algorithm.

6.7. Result Sharing and Randomisation in Normal Form Reduction

Our final topic of this chapter is result sharing between threads and randomisation in normal form reduction. This can be used when reducing results of transactions to normal form using multiple threads, or for parallel reduction of the state to normal form.

Let us assume that we use the `nf` procedure shown, as discussed in Chapter 5, and invoke it with multiple threads on the same node. The problem with this procedure in a concurrent setting is that each thread produces its own structure of Data nodes, meaning that there is no result sharing.

```
1 data Node = ...
2           | Data(constructor : String, children : Node[],
3                  left_to_right : Boolean, is_nf : Boolean)
4           | ...
5
6 nf(root : Node) : Node {
7   whnf_root ← whnf(root);
8   if(whnf_root is DataNode ∧ whnf_root.is_nf = false) {
9     // Reduce children to normal form
10    left_to_right ← ¬left_to_right;
11    if(left_to_right) {
12      for(i ← 0; i < whnf_root.children.length; i ← i + 1) {
13        current ← whnf_root.children[i];
14        current_nf ← nf(current);
15        compareAndSet(whnf_root.children[i], current, current_nf);
16      }
17    } else {
18      for(i ← whnf_root.children.length - 1; i ≥ 0; i ← i - 1) {
19        current ← whnf_root.children[i];
20        current_nf ← nf(current);
21        compareAndSet(whnf_root.children[i], current, current_nf);
22      }
23    }
24    // Mark data node as being in normal form
25    whnf_root.is_nf ← true;
26  }
27  return whnf_root;
28 }
```

Listing 6.10: The `nf` procedure with result sharing and randomisation.

In order to introduce sharing, we use the Data nodes themselves as sharing nodes by updating them instead of constructing new data nodes during reduction. This has the

additional benefit of cleaning up sharing nodes in the graph which result has already been reduced to normal form. A problem now is, how do we know if a data node is in normal form? For weak head normal form reduction, we can simply check if a node is in weak head normal form by reducing it, and see if the reduced node is the same as the original node. For normal form reduction this does not work, as we need to traverse the whole tree to check if there is an unreduced node somewhere in the tree. To solve this, we add a flag to a Data node that encodes if it is in normal form or not. On creation of a data node the flag is not set, and when we reduce a data node to normal form we set the flag.

In order to introduce randomisation, we essentially use the same ideas as for weak head normal form reduction. We alternate the order in which child nodes of a data node are reduced. In our implementation we alternate the order by reducing from left to right, or from right to left, where the order is chosen depending on a flag stored in the data node, as discussed in the previous section.

Listing 6.10 shows the implementation of result sharing and randomisation in the `nf` procedure as described above.

6.8. Conclusions

In this chapter we saw how the graph reducer as discussed in Chapter 5 can be modified to support concurrent and parallel graph reduction. In particular, we have seen:

- The problems in parallel graph reduction.
- Result sharing and randomisation as a method for load balancing the parallel execution of a tree of dependent tasks.
- Using result sharing and randomisation as a method for parallel graph reduction.
- The implementation of result sharing and randomisation in weak head normal form reduction and normal form reduction.

In Chapter 7 we see the application of this graph reducer to the implementation of our transactional functional language. In Chapter 9 we discuss some experiments on the parallel graph reducer to evaluate the relative speedups that can be attained, and how it compares to the serial graph reducer.

7. A Transaction Manager for Transactional Functional Languages

In this chapter we discuss an implementation of a transaction manager for our prototype language. Our transaction manager serves the same purpose as the transaction manager function by Trinder as discussed in Section 3.2. That is, our transaction manager maintains a state and applies a stream of transaction to the state, producing a stream of results. However, our transaction manager differs in two ways from that of Trinder. First, our transaction manager is adapted to the transactional functional language model. Second, we use imperative synchronisation mechanisms to handle concurrent streams of transactions. Additionally, our transaction manager supports the execution of stored transactions. Furthermore, we discuss forcing the evaluation of transactions as a solution to space leaks that show up in the theoretical model due to lazy reduction of states.

In this chapter, we first provide an overview of the implementation, where we discuss the data structures that we use, handling of requests, and setting up the initial state. Next, we discuss the execution of transactions and stored transaction calls in a non-concurrent environment. After that, we discuss how to execute transactions concurrently. Finally, we discuss forcing the evaluation of transactions.

7.1. Overview

First, we provide an overview of the implementation of the transaction manager. We start out by discussing the data structures that we use for the implementation, followed by an overview of the procedures that implement the transaction manager, and finally we briefly discuss setting up an initial state.

Data Structures

First, we discuss the data structures for the transaction processing features of the implementation of the transaction manager. We have data structures to represent a set of definitions, the state, transactions, stored transactions and stored transaction calls.

```
1 type Definitions = Map String Node
```

Listing 7.1: Definitions data structure.

Listing 7.1 shows the data structure for the definitions that are part of a transaction, which is a mapping of identifiers to supercombinators (that may or may not been resolved

yet). A primed identifier denotes that the supercombinator is an update to the state, and a non-primed identifier denotes that the supercombinator is a local definition and must not go into the state. Additionally, an entry $(id, node)$ where $node =$ `null` denotes that the entry id must be removed from the state. Alternatively, it would be possible to use two maps, one for local definitions and one for updates to the state, this is more elegant but complicates the implementation as described in this chapter.

```
1 type StoredTransaction = definitions : Definitions
2                        * parameters : String[]
```

Listing 7.2: Stored transaction data structure.

Listing 7.2 shows the data structure of a stored transaction. A stored transaction consists of a set of definitions together with a list of parameter names.

```
1 type State = data : Map Identifier Node
2            * stored : Map Identifier StoredTransaction
```

Listing 7.3: State data structure.

Listing 7.3 shows the data structure of a state. A state consists of a mapping `data` that maps identifiers to supercombinator templates in the state graph, and a mapping `stored` that maps identifiers to stored transactions.

```
1 type Transaction = definitions : Definitions
2                  * stored_updates : Map Identifier StoredTransaction
```

Listing 7.4: Transaction data structure.

Listing 7.4 shows the data structure of a transaction. A transaction consists of a set of definitions, together with a mapping `stored_updates` that describe updates to the stored transactions in the state. Similar to the Definitions data type, an entry $(id, st) \in$ `stored_updates` where $st =$ `null` denotes that the stored transaction id has to be removed from the state.

```
1 type Call = name : String
2           * arguments : Map Identifier Node
```

Listing 7.5: Call data structure.

Finally, a call of a stored transaction is encoded by the `Call` data structure as shown in Listing 7.5. The `name` field determines which stored transaction is called, and the `arguments` field contains the arguments for the parameters of the stored transaction. Alternatively, a call could be part of a transaction, however this would allow a transaction to have multiple results.

Handling Requests

Now we discuss how transactions and stored transaction calls are handled at a global level.

First, we assume that there is some global variable `state` that contains the state on which transactions are executed. Next, we assume that there is an interface to the outside world on which requests are received to execute transactions, and which returns results to the senders. We assume that this interface invokes either the procedure `handleTransaction` or `handleCall` to respectively handle a transaction or to call a stored transaction. The implementation of these procedures are shown in Listing 7.6.

```
 1 global state : State;
 2
 3 handleTransaction(transaction : Transaction) {
 4   result ← executeTransaction(transaction);
 5   return nf(result);
 6 }
 7
 8 handleCall(call : Call) {
 9   result ← executeCall(transaction);
10   return nf(result);
11 }
```

Listing 7.6: Requests handlers.

To handle a transaction, the `executeTransaction` procedure is invoked. This procedure takes the transaction and the global state, and updates the global state to reflect the state after the execution of the transaction, and produces as a result the root of the result graph of the transaction. The result graph of the transaction is then reduced to normal form, using the procedure `nf` as discussed in Chapter 5.

A stored transaction call is handled similarly, but it invokes the `executeCall` procedure instead. The `executeCall` procedure looks up the stored transaction indicated by the `Call` data structure, and executes this transaction in the current state with the arguments provided by the `Call` data structure.

In the next section, we discuss the implementation of the `executeTransaction` procedure and the `executeCall` procedure. In the section after that, we discuss how concurrent requests can be handled.

Initial State

When the system is first started, an initial state has to be constructed. The initial state should contain the primitive functions that are provided by the system, and may contain other standard definitions.

```
1 initialState() : State {
2   state.put("add", new Add());
3   state.put("sub", new Sub());
4   ...
5 }
```

Listing 7.7: Initialising the state.

As an example, Figure 7.7 shows a procedure that constructs an initial state. The `Add` and `Sub` structures are primitive functions that are part of the graph reduction implementation, as discussed in Chapter 5.

7.2. Executing Transactions and Stored Transaction Calls

Now we discuss the implementation of the `executeTransaction` procedure and the `executeCall` procedure as mentioned in the previous section. For this implementation we do not yet consider the handling of concurrent requests, which will be discussed in the next section.

Executing Transactions

First, we discuss the execution of transactions, which is done as follows. First, free variables in the supercombinators of the transaction are resolved as discussed in Chapter 5. This includes both free variables that refer to supercombinators in the transaction itself, as well as free variables that refer to supercombinators in the state. Next, the updates specified by the transaction are applied to the state, including updates to the supercombinators in the state, as well as updates to the set of stored transactions. Finally, the root of the resolved result supercombinator is returned.

```
1 executeTransaction(Transaction : transaction) : Node {
2   definitions' ← resolve(transaction.definitions, state.data);
3   update(state.data, getUpdates(definitions'));
4   update(state.stored, transaction'.stored);
5   return definitions'.locals.get("result");
6 }
7
8 update(target : Map String a, changes : Map String a) : void {
9   for((key, value) : changes) {
10    if(value = null) {
11      target.remove(key);
12    } else {
13      target.put(key, value);
14    }
15  }
16 }
```

```
getUpdates(environment : Map String Node) : Map String Node {
  updates ← new Map();
  for((var, supercombinator) : resolved) {
    if(var.endsWith("'")) {
      updates.put(
        // Remove the prime from the variable name
        var.substring(0, var.length - 1),
        supercombinator
      );
    }
  }
  return updates;
}
```

Listing 7.8: The `executeTransaction` procedure

Listing 7.8 shows the algorithm for executing a transaction. The `resolve` procedure resolves the free variables in the transaction, of which the implementation is discussed in Chapter 5. Using the `getUpdates` procedure, we filter the updates from the transaction environment and remove the primes from the variable names, to obtain an environment that consists of the resolved updates of the transaction. We update the `data` and `stored` maps through the generic `update` procedure.

Executing Stored Transaction Calls

Using the `executeTransaction` procedure, the implementation of the `executeCall` procedure is relatively straightforward. First we fetch the stored transaction as indicated by the call. Then we insert the arguments into this transaction as if they were locally defined supercombinators. Finally, we execute this transaction using the `executeTransaction` procedure.

```
executeCall(call : Call) : Node {
  stored_transaction ← state.stored.get(call.stored);
  if(stored_transaction = null) {
    throw new Exception("Stored transaction does not exist");
  }
  if(stored_transaction.parameters.keySet() ≠ call.arguments.keySet()) {
    throw new Exception("Arguments do not match parameters");
  }
  definitions ← stored_transaction.definitions.putAll(call.arguments);
  return executeTransaction(new Transaction(definitions, new Map()));
}
```

Listing 7.9: The `executeCall` procedure.

Listing 7.9 shows the implementation of the `executeCall` procedure. Here we perform some additional checks to ensure that the stored transaction actually exists, and that

the arguments passed to it match the parameters. If this last check where to be omitted, local definitions of the transaction could be overwritten, creating a potential security leak. We assume that the `putAll` function does not modify the stored transaction, but that it returns a new transaction with the arguments added to it. Now the reason why `resolve` has been implemented non-destructively should also be clear, because if it would update the expression to resolve it, the stored transaction can only be used once.

7.3. Handling Concurrent Transactions

Now we discuss how requests can be handled concurrently. In order to handle concurrent requests, we assume that the interface to the outside world may create multiple threads that may invoke the `executeTransaction` procedure and the `executeCall` procedure concurrently. The implementation for these procedures as discussed in the previous section are not correct in such a concurrent environment, as the operations on the state may interfere, leading to race conditions.

```
1 handleTransaction(transaction : Transaction) {
2   synchronized(state) {
3     result ← executeTransaction(transaction);
4   }
5   return nf(result);
6 }
7
8 handleCall(call : Call) {
9   synchronized(state) {
10    result ← executeCall(transaction);
11  }
12  return nf(result);
13 }
```

Listing 7.10: Handling requests concurrently using locks.

The simplest method to ensure correct updating of the state is to make access to the state mutually exclusive. This can be implemented using a lock in the request handler procedures as shown in Listing 7.10. The construct `synchronized(state)` ensures mutual exclusive access to its code block. Normal form reduction does not have to be part of the mutual exclusive region, as it may be done concurrently using the parallel graph reducer as discussed in Chapter 6.

However, we can do better than this. As read transactions do not modify the state, these can be executed concurrently. This can be implemented using a readers / writers lock, where a write lock is chosen if the transaction wants to perform updates, otherwise using a read lock. However, we can do even better than that if we can atomically update the state, as this allows reads to be lockless. We can make updates to the state atomic

by performing updates on the state non-destructively to obtain a new state next to the existing state, and then updating the global `state` variable atomically to point to this new state.

```
executeTransaction(transaction : Transaction) : Node {
  if(getUpdates(transaction.definitions).size() = 0
     ∧ transaction.stored.size() = 0) {
    definitions' ← resolve(transaction.definitions, state.data);
  } else {
    synchronize(state) {
      definitions' ← resolve(transaction.definitions, state.data);
      state' ← new State();
      state'.data ← update(state.data, getUpdates(definitions'));
      state'.stored ← update(state.stored, definitions'.stored);
      state ← state';
    }
  }
  return definitions'.locals.get("result");
}
```

Listing 7.11: Lockless reads for transactions.

Listing 7.11 shows the implementation of this approach to the execution of transactions. We first check if the transaction performs any updates. If the transaction does perform updates, we enter a mutually exclusive region and we construct a new state with the updates. We update the global `state` pointer to point to our newly constructed state to update the state atomically. The `update` function now constructs a new mapping, instead of updating the existing ones. If the transaction does not perform any updates, we take the current state and resolve free variables in the transaction. Once we got a pointer to a state data structure, that data structure does not change anymore, as updates construct a new state instead of modifying the current state.

Listing 7.12 shows the implementation for stored procedure calls. In order to execute stored transactions, we first get the state and store it in a local variable `local_state`. We then look up the stored transaction from this state, and check if this stored transaction updates the state. If the stored transaction does not update the state, we execute it on the state that we stored in `local_state`. If the stored transaction does update the state, we enter a mutually exclusive region. Between fetching the state and going into the stored region, another transaction may have updated the state and modified the stored transaction, so we have to fetch a new copy. We then proceed by executing this stored transaction as shown earlier.

```
1 executeCall(call : Call) : Node {
2   local_state ← state;
3   st ← local_state.stored.get(call.stored);
4   if(getUpdates(transaction).size() = 0) {
5     definitions ← st.definitions.putAll(call.arguments);
6     defintiions' ← resolve(definitions, local_state.data);
7   } else {
8     synchronized(this) {
9       st ← state.stored.get(call.stored);
10      definitions ← st.definitions.putAll(call.arguments);
11      definitions' ← resolve(definitions, local_state.data);
12      state' ← new State();
13      state'.data ← update(state.data, getUpdates(definitions'));
14      state'.stored ← update(state.stored, definitions'.stored);
15      state ← state';
16    }
17  }
18  return definitions'.locals.get("result");
19 }
```

Listing 7.12: Lockless reads for stored procedure calls.

7.4. Forcing Evaluation of Transactions

A problem when reducing states lazily is that we may get long chains of redexes in states as a result of a chain of transactions that update the state but which do not read the state. In a practical implementation, this can lead to a stack overflow when reading after many updates.

Additionally, during our experiments we also found that space leaks may build up in the state, as discussed in Section 9.3. Our experiments shows that a `map` function applied to a binary tree may leave redexes applied to `Leaf` nodes, which result may never be demanded by transactions that read the state, causing a space leak.

Another related problem is that sharing nodes are created in the state, but these are not cleaned up when they are not needed anymore. This happens because the reduction of result expressions forces reduction in the state, which may create result sharing nodes, but which can not clean up these result sharing nodes.

Our initial solution to these problems was to force the reduction of states to normal form, as this forces the reduction of all redexes. Also, we limit the number of active transactions to avoid a build-up of a long chain of redexes before we can reduce them. In order to implement this, we keep track of which parts of the state have already been reduced to normal form, in order to avoid duplicate reduction of the state every time its

reduction is forced. We have already implemented this in 6.7, by maintaining a flag on `Data` nodes to indicate whether it is in normal form.

However, we found that simply forcing the reduction of states to normal form does not work as intended for limiting the number of active transactions in a concurrent setting. The problem is that if a small transaction is preceded by a very large transaction, and we force the evaluation of the state created by the small transaction, this also forces the reduction of the state produced by the large transaction. Because of that, we can not know accurately when the reduction of the small transaction is done, which is needed to count the number of active transactions.

A theoretical solution to this problem is to reduce only those parts of the state that a transaction modifies, i.e. forcing the evaluation of a transaction. To implement this, for all redexes, we need to keep track of which transaction it belongs to. In order to force the evaluation of a transaction, we need to force the reduction of all redexes that belong to that transaction. We plan to further investigate the implementation of this solution.

7.5. Conclusions

In this chapter we have discussed the implementation of a transaction manager for transactional functional languages. In particular we have seen:

- Data structures that can be used for the implementation of a functional transaction manager, and a global overview of the execution of a transaction.
- Algorithms for the execution of transactions and the execution of calls to stored transactions.
- An algorithm for the concurrent execution of transactions that allows lockless reads.
- The problem caused by lazy reduction of states, and a solution based on forcing the evaluation of transaction.

With what we have seen so far we can already implement a complete functional transaction processing languages. However, we have not discussed persistence yet, this is the topic of the next chapter. Furthermore, in Chapter 9 we asses the performance of the locking and lockless reads approach to executing transactions concurrently.

8. Maintaining Persistence

So far we have seen the implementation of graph reduction, and a transaction manager for the implementation of a transactional functional language. In this section we discuss different ways to efficiently store the state in persistent memory, with the goal of guaranteeing durability in case of a system failure, storing reduction results, and to be able to work with states that do not fit in main memory.

In this chapter, we first discuss the characteristics of persistent storage media. Next, we discuss journaling as a method to guarantee durability. After that, we discuss snapshotting as a method to store reduction results, as well as being able to store results of ongoing computations. Following that, we discuss log-structured storage as a method to work with states that do not fit in main memory. Next, we combine the snapshotting and the log-structured storage approach to combine their strengths, while minimising their weaknesses. Finally, we discuss the implementation of journaling and snapshotting for our prototype implementation.

8.1. Characteristics of Persistent Storage

Up to now we have assumed that the state is stored in non-persistent main memory. We define *persistent memory* as memory that retains its contents when the system restarts after a failure or reboot, whereas *non-persistent memory* loses its contents when the system restarts.

Current implementations of persistent memory can be found mainly in the form of hard disk drives and flash drives. Hard disk drives are the cheapest per unit of storage, but they suffer from high latency to access data, which is around 10 milliseconds. Solid state drives are currently about 10 times as expensive as hard drives, but they have significant lower latency at about 0.1 milliseconds. Compared to current non-persistent memory with access times in the nanoseconds, current persistent storage mechanisms are very slow. This means that implementing graph reduction directly in persistent memory is not a realistic option.

In persistent memory, reading and writing data sequentially is generally comparatively fast compared to random access. To reason about the performance of a system working with storage with slow access times, we can reason in terms of the number of *I/O operations* required to execute a task, where one I/O operation is a sequential read or write. Generally, we try to minimise the numbers of I/O operations, by performing more useful work per I/O operation. This can be achieved by putting elements of data close

to each other that commonly need to accessed together, and reading elements of data in blocks so that many relevant elements are read in a single I/O operation. Essentially, we try to maximise the *locality of reference* of data.

8.2. Journaling

Journaling [19] is a standard method in databases for guaranteeing durability of transactions in the event of system failure.

The idea of journaling is that a transaction is written to a *transaction log* prior to its execution. If the system crashes and starts up again, it *recovers* the state by cleaning up the effects of partially executed transactions, and then re-executing all transactions to obtain the state as it was before the crash. To guarantee durability to the user, the system must ensure that a log entry is actually written to persistent memory before confirming the execution of the transaction to the client.

In theory, having an initial state and a journal starting in this initial state provides enough information to reconstruct the state at any point in time. However, in practice this is not really sufficient: the log grows beyond bounds as entries can never be removed, and moreover it is extremely inefficient to re-execute all transactions when a journal grows large. For this reason, we want to store reduced forms of states as a *checkpoint*, so that the system only has to recover from the last checkpoint. In the next section we discuss snapshotting, which is a method for creating such checkpoints. We discuss the implementation of journaling later in this chapter.

8.3. Snapshotting

We now discuss snapshotting as a method to create a checkpoint for journaling. The idea of *snapshotting* is that we serialise the state, and write it to persistent memory. To restore the state, we can simply deserialise the state from persistent memory to obtain the original state.

We assume that sharing is maintained when serialising and deserialising the state. The maintenance of sharing is usually implemented by remembering the positions of nodes that have already been written. If the serialisation process encounters a node that has already been written, it writes a pointer to the existing node instead of writing a copy of the node. However, this means that serialisation requires memory linear in the number of nodes in the graph that is serialised. Further details about the implementation of serialisation are outside the scope of this thesis.

Snapshotting may seem like a simple task, but there are some complications if we want to snapshot states containing redexes while concurrently reducing the state.

The problem with concurrent reduction during snapshotting is that sharing may be lost in a snapshot. Consider the example in Figure 8.1. The serialisation process starts by serialising the root node `Cons`, and then proceeds by recursively serialising the left and

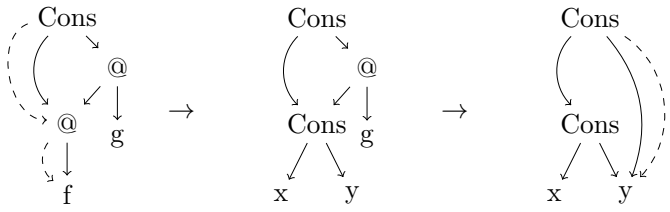


Figure 8.1.: Concurrent serialisation and snapshotting.

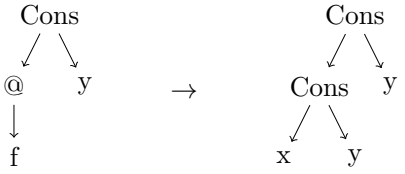


Figure 8.2.: Loss of sharing in the snapshot.

right branches of the root node. First, the serialisation process serialises the left branch, as shown by the dashed line in the leftmost state. Now, another thread may become active and reduce the graph to obtain the rightmost state. After that the serialisation process continues by serialising the right branch of the root node.

The snapshot obtained by the snapshotting process is shown on the left side in the Figure 8.2. When this snapshot is reduced, we obtain the graph on the right side, and we find that sharing of the value `y` is lost. This example effectively shows that sharing may be lost when snapshotting is performed concurrently with reduction.

As we do not want to block access to the state when creating a snapshot, a simple solution to solve the problem of losing sharing is to reduce the state to normal form before snapshotting it. This is also the approach that we have implemented in our prototype, as discussed later in this chapter. However, the drawback of this approach is that snapshotting could be delayed for a long time while a large update is being evaluated. This means that in the mean time, the journal can grow very large, leading to a long recovery time if there is a system failure before snapshotting is finished.

Ideally, we want to be able to take a snapshot without reducing the state first, i.e. we want to be able to snapshot ongoing computations. To do this, we need to make it seem as if the snapshot is created while no reduction was going on. We observe that the only nodes reachable from the root that change during reduction in our graph reducer are the sharing nodes. The essential idea to create a consistent snapshot is to make a reduction thread snapshot the state of a sharing node before updating it while snapshotting is going on. In order to implement this, we can remember the original shared node of a sharing node when it is reduced during the creation of a snapshot. If the snapshot process encounters a sharing node, we see if there is an original shared node stored for

this sharing node, and if so we snapshot the original shared node instead of the current shared node.

One complication with this approach is that there is no garbage collection for graph reduction at all during snapshotting, as we keep a reference to each original node in the graph. We can solve this by only remembering the original reference for sharing nodes that have been constructed before snapshotting was started. To do this, we need to know if a node was created before the start of the snapshot, or if it has been created during the snapshot. This can be implemented by sequentially numbering the snapshots, and by assigning each sharing node a number indicating to which snapshot it belongs. When snapshotting is started, a global snapshot number is incremented by one, and only sharing nodes constructed from there on have this incremented snapshot number. If a thread reduces a node with the current snapshot number, it does not have to store the original shared node. When the snapshot procedure has snapshotted a node, it can increment the snapshot count of that node by one to ensure that the original shared node is no longer stored by reduction threads for that node. We have not implemented this approach in our prototype, and additional details remain a topic of further investigation.

The main advantages of snapshotting compared to the approach that we see in the next section is that we can create checkpoints of ongoing computation, and that sharing is maintained in snapshots. The biggest drawback of snapshotting is that we can not support states that are larger than main-memory. Another drawback is that snapshotting can take considerable time to complete. If there is a high load on the system, we might have to snapshot quite often to keep the journals small enough for quick recovery times. As the size of the state grows, taking snapshots takes a longer time, and the journals grow larger between every subsequent snapshot, leading to longer recovery times.

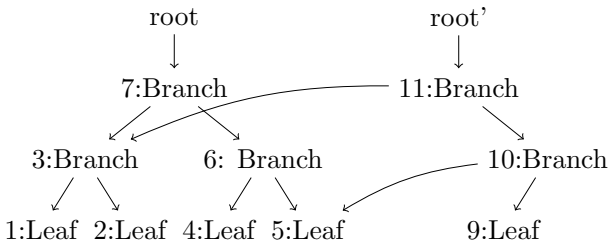
8.4. Log-Structured Storage

In the snapshotting approach, large parts of the state might not have changed between snapshots, so creating a completely new snapshot is inefficient. An alternative approach to store data that solves this problem is *log-structured storage* [22], also known as *append-only storage*.

The main idea of log-structured storage is that the nodes in the state are stored as records in a log file, where records can point to other records by their position in the file to encode graph edges. After the execution of a transaction, all new nodes in the state are *appended* to the log, while they may refer to nodes that have already been written to the log. After all nodes have been written, a special root record is written that encodes the bindings in the state after the transaction. This storage model fits the functional model well, as it is inherently non-destructive.

An advantage of log-structured storage is that data is only appended, and never overwritten. If the system crashes while appending records to a log, there is no risk that

data that has already been written to the log is corrupted. State recovery is simply a matter of finding the last correctly written root record from the end of the log file, and reading the state by following the pointers. One can consider this approach as a continuous snapshot of the system state in a single file. This has the advantage that we always have an up-to-date snapshot of the system by appending only the changes since the last snapshot, instead of writing the whole state every time.




```
 1 Leaf
 2 Leaf
 3 Branch  1 2
 4 Leaf
 5 Leaf
 6 Branch  4 5
 7 Branch  3 6
 8 Root  7
 9 Leaf
10 Branch  9 5
11 Branch  3 10
12 Root  12
```

Figure 8.3.: Log-structured storage example.

Figure 8.3 illustrates how log-structured storage works through an example. On the left side we see two subsequent states, indicated by `root` and `root'`, which reside in main memory. On the right hand side we see what is written to the log to store these states. We see that each entry in the log has a position by which it can be referred, which is in this case its line number, but in practice the byte offset from the beginning the log file can be used. First, `root` is written to the log file. We see that nodes in the log file refer to one another by their location in the log. For every node in the graph, we remember at which location in the log it is written, so that we can refer to it later. When `root'` is written to the log, only the new nodes are written to the log, while referring to the nodes that have already been written by their location in the log.

One of the main advantages of log-structured storage is that it allows states that are larger than main-memory. This can be achieved by dynamically *unloading* parts of the graph that have been written to the log file, and replacing these parts by a special node that marks where its contents can be found in the log file. If we encounter such a special node during reduction, we can dynamically load that part of the graph into memory. An illustration of how this approach works can be seen in Figure 8.4. When during reduction we encounter the node `Stored`, we load the associated graph from the log and replace `Stored` by this graph. The dynamic loading and unloading of parts of the graph could be implemented using a caching mechanism, using a least-recently used policy

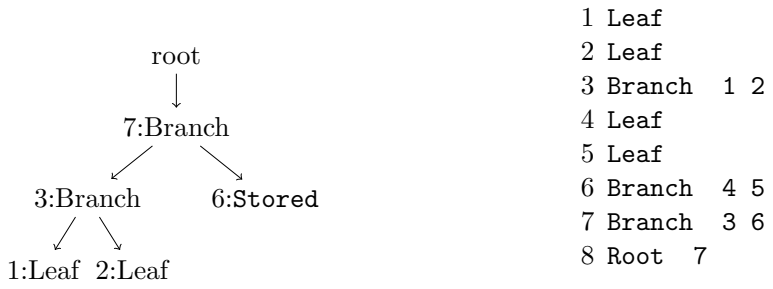


Figure 8.4.: Graph with an unloaded branch.

to determine which nodes to unload. That is, when the system is low on memory, it could unload the top n least-recently used nodes. However, we have not investigated the implementation of this, so this remains a topic of further research.

A main concern when dynamic loading parts of the graph is that this may be very slow because many I/O operations may be needed to read the relevant parts from the log file. To minimise the number of I/O operations needed, we should maximise the amount of relevant nodes read per I/O operation. To do this, we can write parts of the graph in blocks, e.g. the size of a disk page, where blocks are filled using breadth-first traversal of the graph. When a node is to be loaded from the log, we also get all nodes that are in the same block, which are likely to be relevant.

A major complication of the log-structured approach is that the log only grows, and never shrinks in size. Further, the layout of the log becomes sub-optimal as changes to the state are appended to the log. Both of these problems can be solved simultaneously by using garbage collection to periodically clean up the log, while ensuring that the cleaned up log is written in blocks as described in the previous paragraph. A problem in garbage collection is that it is very expensive to maintain sharing in a cleaned up log file. This is because we have to remember which nodes have already been seen. If the log is very large, this may requires more memory than available. If an application of the system does not depend on sharing for memory efficiency, this should however be no problem. The efficient implementation of garbage collection in this setting is a topic of further investigation.

Further, log-structured storage is inefficient for storing suspended computations. If we store a redex, and later compute its result, we have to somehow ensure that the log is updated to contain this result. We can not actually overwrite the result in the log in-place, because if the system crashes while updating a record, the log may be corrupted due to incomplete written records. Instead, we could append every path in the graph from the root to the updated result. However, there may exist many paths to the result, and as we cannot see locally which expressions reference the redex, it can be expensive

to find all these paths. Furthermore, writing all these paths can be require quite a lot of space. In the next section, we discuss an alternative solution to this problem.

Finally, in our current language model, the root of the graph is the environment that maps identifiers to expressions. If this environment contains many entries, writing it to the log for every transaction becomes very expensive. A potential solution for this to store this environment as a tree structure, such that we can refer to parts of previously written roots when writing a new root. However, in the next section we see an alternative approach to solve this problem.

8.5. Mixed Approach

We now combine both the snapshotting approach and the log-structured approach, to obtain an approach that allows both storage of suspended computations, as well as allowing states that are larger than main-memory, while providing low recovery times.

The idea is that we split the heap into an *active heap* which may contain reducible expressions, and a *passive heap* which must be in normal form. When reducing nodes in the active heap to normal form, these nodes are moved to the passive heap.

As the passive heap is in normal form, it can be stored efficiently using log-structured storage. This allows the passive heap to be larger than main-memory. However, this also means that we can not ensure that sharing is maintained due to garbage collection. But as there are no suspensions in this heap, this affects only the amount of memory required to store the state, and it does not affect the efficiency of lazy evaluation.

For the active heap we use the snapshotting approach, as to allow checkpointing of long-running transactions. Instead of using special root nodes in the passive heap, the active heap serves as the root of the passive heap. This means that when a snapshot is created, we have to ensure that the referenced nodes in the passive nodes have all been written to the log. Recovery is a matter of restoring the active heap, which can be done by loading the last snapshot and re-applying the transactions in the journal since the last snapshot.

We have to ensure that the active heap stays small, so that it can fit in main memory. To do this, we can force the evaluation of states, such that they can be written to the passive heap as quickly as possible. For this, the technique as described in Section 7.4 can be used.

If we can manage that the active heap stays small, it is possible to create snapshots much faster than in an approach using only snapshotting. Being able to perform faster snapshots allows us to perform more snapshots. This means that the number of new journal entries between snapshots is smaller, leading to lower recovery times than the snapshotting-only approach.

8.6. Implementing Journaling and Snapshotting

In our prototype implementation we have implemented journaling together with the snapshotting approach. To avoid the sharing issues of snapshotting as discussed in Section 8.3 we reduce states to normal form before snapshotting.

Preliminaries

To work with persistent memory, we assume the existence of a simple file system that provides the following procedures:

`openFile` : **String** $\rightarrow$ **File**
: Given the name of a file, returns a file handle for reading and writing to the file.

`closeFile` : **File**
: Given a file handle, closes a file.

`serialise` : $a \times$ **File**
: Takes a data structure of some type a, and converts this to a steam of data and appends this to the given file.

`deserialise` : **File** $\times a$
: Reads a stream of data from a file and converts this to a data structure of some type a. If the end of the file is reached before or during deserialisation, the procedure returns `null`.

`createFile` : **String**
: Given the name of a file, creates that file.

`deleteFile` : **String**
: Given the name of a file, deletes that file.

`renameFile` : **String** $\times$ **String**
: Given the current name of a file and a new name, renames the file from the current name to the new name.

We assume that creating, deleting and renaming files are operations that are both atomic and durable. That is, we assume that when we invoke such an operation on the file system, it has actually been performed after control is returned to the program, and its effects persist in the event of system failure. If the system crashes during such an operation, and control has not been returned to us, then we assume that the operation has either been executed completely, or it has not been executed at all.

We assume that appending to files is durable, but not necessarily atomic. That is, when control is returned after invoking an append operation, we assume that the data has been written to the file, and the effects of the append operation persists in the event of a system failure. If there is a system failure during an append operation, we assume that some data may have been written, but not necessarily that all data has been written. Furthermore, we assume that there is no corruption of data in case of system failure.

That is, append operations that have been executed successfully are not affected by a system failure.

In our implementation we use the serialisation mechanism provided by Java to implement `serialise` and `deserialise`.

Journaling

Now we discuss the implementation of journaling. We first discuss writing transactions to journals, followed by recovering the state by applying the transactions in a journal to the state.

To write a transaction to a journal, we append a serialised version of the transaction to the journal file before binding free variables. Additionally, transactions have to be journaled in the same order as in which they have been applied to the state, in order to guarantee that we obtain the same state after recovery as that we have before recovery. As our transaction manager uses mutual exclusion to ensure that concurrent update transactions are applied serially, we can perform journaling in this mutual exclusive region to guarantee that journals are written in the same order as that the transactions are applied.

```
1 global journal : File
2
3 executeTransaction(transaction : Transaction) : State {
4   ...
5   synchronize(state) {
6     // Journal the transaction
7     serialise(transaction, journal);
8
9     // Execute the transaction
10    ...
11  }
12  ...
13 }
```

Listing 8.1: Journaling transactions.

Listing 8.1 shows how journaling is implemented for the execution of transactions, extending the `executeTransaction` procedure as discussed in Chapter 7. In this implementation we assume that there is a global file handle `journal` to which log entries can be written. The implementation of journaling for stored transaction calls is similar.

The drawback of this implementation is that each transaction performs an I/O operation, which limits throughput. An approach to reduce the number of I/O operations is to buffer incoming transactions while an I/O operation is in progress. All buffered transactions can be written sequentially using one I/O operation once the initial I/O

operation completes, followed by executing the transactions in the same order as that they where written. However, the implementation of this is future work.

To recover the state from a journal, we deserialise the transactions in the journal and apply them to the state in the same order as that they have been written. An entry in a journal might have been corrupted if there was a system failure while appending to a journal. If `deserialise` encounters an incomplete or corrupted entry, it returns `null`. Instead of attempting to repair a log file, we use a new log file every time the system is restarted, while keeping the existing log files.

```
1 applyJournal(file : File) : Void {
2   transaction ← deserialise(file);
3   while(transaction ≠ null) {
4     execute(transaction);
5     transaction ← deserialise(file);
6   }
7 }
```

Listing 8.2: The `applyJournal` procedure.

Listing 8.2 shows how recovering a journal is implemented. Given a journal file handle, this procedure updates the state by the applying the transactions stored in the journal. We assume that `deserialise` returns `null` when the end of file is reached, or if an incomplete serialised entry is encountered. For simplicity, we assume that the `execute` procedure distinguishes between regular transactions and stored transaction calls.

Snapshotting

Now we discuss our implementation of snapshotting.

As discussed in the previous sub-section, every time the system recovers, we create a new journal. To ensure that journal files are cleaned up correctly, we associate a sequence of journals $\{n, \ldots, m\}$ with a snapshot. As soon as a snapshot has been successfully written to persistent memory, its associated journal files are deleted, as they are not needed anymore. Additionally, cleaning up journal files may fail if there is a system failure while cleaning up journal files. For this reason, we store the journal files associated to a snapshot in the snapshot, as this allows us to clean up any remaining journal files during recovery.

```
1 global firstJournal : Integer
2 global lastJournal : Integer
3 global sumJournalSize : Integer
4 global constant snapshotThreshold : Integer
5
6 data Snapshot = state : State * firstJournal : Integer * lastJournal :
      Integer
```

Listing 8.3: Globals for snapshotting.

Listing 8.3 shows the global state and a data structure that we use to keep track of the journals associated to a snapshot. The variables `firstJournal` and `lastJournal` respectively encode the begin and end of the sequence of journal numbers associated to the next snapshot. The variable `sumJournalSize` keeps track of the size of all journals associated to the next snapshot that have been closed for writing. The `snapshotThreshold` is a constant that determines when the snapshotting procedure is started. The `Snapshot` data structure is used to serialise a snapshot to disk, and keeps track of the journals associated to this snapshot and the snapshotted state.

```
1 global snapshotting : Boolean
2
3 executeTransaction(transaction : Transaction) : State {
4   ...
5   synchronize(state) {
6     // Journal the transaction
7     serialise(transaction, journal);
8
9     // Start snapshotting
10    if(sumJournalSize + size(journal) > snapshotThreshold) {
11      startSnapshot();
12    }
13
14    // Execute the transaction
15    ...
16  }
17  ...
18 }
```

Listing 8.4: Initialising a snapshot.

To initialise a snapshot, we extend the `executeTransaction` and `executeCall` procedures as shown in Listing 8.4. When the cumulative size `sumJournalSize` of all the journal files associated to the next snapshot plus the size of the current journal `size(journal)` grows beyond `snapshotThreshold`, then the snapshotting procedure is started by invoking `startSnapshot`.

In the `startSnapshot` procedure, shown in Listing 8.5, we first check if there is already a snapshot in progress, if so we abort the procedure. Next, we set the `snapshotting` flag to mark that snapshotting is in progress. We assume that the procedure is invoked in a mutual exclusive setting, otherwise there is a data race because we check and update the `snapshotting` flag non-atomically. Next, we spawn a new thread that actually performs the snapshotting by invoking the `snapshot` procedure with the current state and the journal sequence numbers associated to the snapshot. The `snapshot` procedure is discussed below. A new thread is used, as to not block the transaction that invoked the `snapshot` procedure. When snapshotting has been started, all new transactions are

part of the next snapshot. We set up the next snapshot by creating a new journal, associating only that journal to the next snapshot, and resetting `sumJournalSize` to zero.

```
1 startSnapshot() : Void {
2   if(¬snapshotting) {
3     snapshotting ← true;
4
5     // Start the snapshotting procedure
6     spawn thread: snapshot(state, firstJournal, lastJournal);
7
8     // Create a new journal
9     lastJournal ← lastJournal + 1;
10    firstJournal ← lastJournal;
11
12    closeFile(journal);
13    createFile("journal." + lastJournal);
14    journal ← openFile("journal." + lastJournal);
15
16    sumJournalSize ← 0;
17  }
18 }
```

Listing 8.5: Starting a snapshot.

```
1 snapshot(state : State, firstJournal : Integer, lastJournal : Integer) :
      Void {
2   state ← nf(state);
3
4   snapshotFile ← openFile("new_snapshot");
5   snapshot ← new Snapshot(state, firstJournal, lastJournal);
6   serialise(snapshot, snapshotFile);
7   closeFile(snapshotFile);
8
9   renameFile("new_snapshot", "snapshot");
10
11  // Delete old journals
12  for(i ∈ [firstJournal .. lastJournal])
13    deleteFile("journal." + i);
14  }
15
16  snapshotting ← false;
17 }
```

Listing 8.6: The `snapshot` procedure.

The actual creation of a snapshot is performed by the `snapshot` procedure, as shown in Listing 8.6. The `snapshot` procedure takes as input a state and its associated journal numbers {`firstJournal`, ..., `lastJournal`}. First, the state is reduced to normal form, to avoid problems with snapshotting while the state is concurrently being reduced by other threads, as discussed in Section 8.3. Next, we wrap the reduced state together with its associated journal file numbers into a `Snapshot` data structure. We serialise the snapshot, and write it to a file `new_snapshot`. We rename `new_snapshot` to `snapshot` when snapshotting is done, this ensures that the last correctly written snapshot is not corrupted if there is a system failure during snapshotting. When the snapshot has been written, we delete the journal files associated to the snapshot that has just been written, as they are not needed anymore. Finally, we set `snapshotting` to `false` so that the next snapshot procedure can start.

Recovery

We now discuss the implementation of the recovery procedure, which is invoked when the system starts.

The `recover` procedure recovers the state of the system using the journal files and log files, and ensures that old journal files are cleaned up. During recovery, we may encounter four cases:

- The file `snapshot` does not exist, and the file `new_snapshot` does not exist: we assume that there is no stored state, so we set up a new state.
- The file `snapshot` exists, and the file `new_snapshot` does not exist: we can load `snapshot` and apply any new journals.
- The file `snapshot` exists, and the file `new_snapshot` also exists: There was a system failure during the creation of a new snapshot, we can delete `new_snapshot`, load `snapshot`, apply the new journals, and restart the snapshot procedure.
- The file `snapshot` does not exists, and the file `new_snapshot` does exist: `new_snapshot` contains a correct snapshot, but the there was a system failure while renaming `new_snapshot` to `snapshot`, so we can just rename `new_snapshot` to `snapshot`.

```
1 recover() : Void {
2   snapshotting ← false;
3
4   if(¬file_exists("snapshot"))
5     if(¬file_exists("new_snapshot")) {
6       firstJournal ← 0;
7       lastJournal ← 0;
8       sumJournalSize ← 0;
9       create_file("snapshot");
10      state = initialState();
11    } else {
12      renameFile("new_snapshot", "snapshot");
13      load();
14    }
15  } else {
16    load();
17    if(file_exists("new_snapshot")) {
18      startSnapshot();
19    }
20  }
21
22  // Create a new journal
23  createFile("journal." + lastJournal);
24  journal ← openFile("journal." + lastJournal);
25 }
```

Listing 8.7: The `recover` procedure.

Listing 8.7 shows how the recovery procedure is implemented. To set up a new state, we initialise the journal sequence numbers to zero, we create a dummy `snapshot` file, so that if the system fails before the first snapshot is created, the recovery procedure will restore the state from the journals instead of again creating a new state. An implementation of `initialState` for setting up the initial state is discussed in Section 7.1. To load a state, we use the `load` procedure, as discussed later in this section. At the end of the recovery procedure we create a new journal file, where we assume that `lastJournal` already contains the sequence number of this new journal file.

```
1 load() : Void {
2   file ← openFile("snapshot");
3   snapshot ← deserialise(file);
4   closeFile(file);
5
6   state ← snapshot.state;
7
8   // Delete old journals
9   for(i ∈ [snapshot.firstJournal .. snapshot.lastJournal])
10    deleteFile("journal." + i);
11  }
12
13  firstJournal ← snapshot.lastJournal + 1;
14  lastJournal ← firstJournal;
15
16  // Apply new journals
17  file ← openFile("journal." + lastJournal);
18  while(file ≠ null) {
19    applyJournal(file);
20    sumJournalSize ← sumJournalSize + size(file);
21    closeFile(file);
22    lastJournal ← lastJournal + 1;
23    file ← openFile("journal." + lastJournal);
24  }
25  closeFile(file);
26 }
```

Listing 8.8: The `load` procedure.

The `load` procedure shown in Listing 8.8 loads a state from a snapshot file and applies new journal files. During the creation of the snapshot, there might have been a system failure after creating the snapshot, but before deleting the old journals. To be sure that the journals are correctly cleaned up, we delete them again. Finally, we then check if there are any new journals and apply them.

8.7. Conclusions

In this chapter we have seen methods for the storage of functional states in persistent memory, in particular we have seen:

- Characteristics of persistent storage media.
- Journaling as a method to guarantee durability.
- Snapshotting as a method to create journaling checkpoints, supporting checkpointing of suspended computations.
- Log-structured storage as a method to storage states larger than main-memory.
- A mixed approach of snapshotting and log-structured storage, allowing both checkpointing of suspended computations, as well as allowing large states, while allowing low recovery times.
- Our implementation of journaling and snapshotting for our language prototype.

This chapter concludes the implementation of our prototype. In the next chapter we continue with the experimental validation of our prototype.

Part III.

Evaluation

9. Experiments

Using our prototype implementation, we have performed several experiments to asses the performance of our graph reduction method, as well as the performance of the prototype implementation for transaction processing. The transaction processing experiments are divided into two parts, one investigating the behaviour of our prototype under concurrency, and one investigating the throughput performance under high load. First, we discuss the experimental setup of our experiments in general, then we discuss our parallel graph reduction experiments, followed by our concurrency experiments, and finally we discuss our throughput experiments.

9.1. Experimental Setup

To evaluate our prototype, we run experiments to asses the performance of our parallel graph reducer, as well as its transaction processing capabilities. The precise details of our experimental setup are described per experiment in the sections following.

We do not test the performance of our on-disk persistence mechanism, as the implementation is not yet suitable for high-performance transaction processing. Instead, all benchmarks are run on an instance that resides completely in system memory.

For all experiments we embed the persistent language in our testing program as to eliminate network overhead. This means that, in the transaction processing benchmarks, all transactions are issued from the same process that also contains our persistent language. As to give an idea how this works, the interface used to communicate with the persistent language is a method `String execute(String transaction)`, that accepts a transaction encoded in our language and returns a string containing the normal form of the result of the executed transaction.

In general, for all benchmarks the Java Virtual Machine (JVM) is first warmed up by running a few benchmarks before the measurements start. This warming up triggers the just-in-time compilation of the JVM such that native machine code is generated, instead of the code being interpreted. If we would omit this step, the just-in-time compiler would trigger during measurements, which leads to inconsistent results.

All our experiments have been conducted on a quad AMD Opteron 6168 system that has 48 cores divided over four processors that each have two NUMA nodes. The system has a non-uniform memory access (NUMA) architecture, meaning that each NUMA node has its own local memory. All memory can be accessed from any NUMA node, however it takes more time to access memory that is local to another NUMA node than to access

the memory associated to the local NUMA node. This means we can run benchmarks using up to twelve cores to measure parallel performance on a single processor, and we can run benchmarks using up to 48 cores to measure the performance on a NUMA architecture.

The operating system used is scientific linux, running Oracle HotSpot JVM version 1.7.0 build 147. All benchmarks have been run with the -server option, using the default throughput garbage collector. Due to technical issues, we were unable to use the `-XX:+UseNUMA` option on our test system, which would enable NUMA aware memory allocation.

9.2. Parallel Graph Reduction

In this section, we discuss our experiments to asses the performance of our parallel graph reducer. First, we describe our method for assessing the performance. Next, we present and discuss the results of our experiments. We conclude this section with a discussion about the strengths and weaknesses of our parallel graph reducer.

Setup

In order to asses the performance of our parallel graph reduction method, we have implemented two programs on which we run our benchmarks:

- `nfib` This program computes the nth Fibonacci number. The implementation is done naively, for the purpose of measuring parallel performance. This program is highly parallel, and is not data intensive.
- `treesize` We build a balanced binary tree of n levels deep, and then query for the size of this tree, where we measure the time for querying the size of the tree. This program is also highly parallel, but is much more memory intensive than `nfib`.

For each algorithm, we have implemented two variants. One is written in our prototype language, and one implementation is written as a primitive function that is implemented in Java. We refer to the second implementation as the 'native' implementation. The reason for implementing the algorithms natively as well, is that this allows us to simulate the performance of our graph reduction method for a compiled functional language.

We have run two different sets of benchmarks. First, we measured the relative speedup by executing the algorithm with one up to 48 threads, and dividing the measured execution time with the execution time when using a single thread. Second, we measured the overhead of our parallel graph reducer compared to our serial graph reducer.

During benchmarking the execution time of a program may differ between runs. Causes for this include the non-determinism from multiple threads interacting, the Java garbage collector, as well as background tasks on the testing system. To compensate for this in our benchmarks, we execute each configuration multiple times, and take the median of the execution time. We chose to use the median instead of the average as this is more robust against outliers.

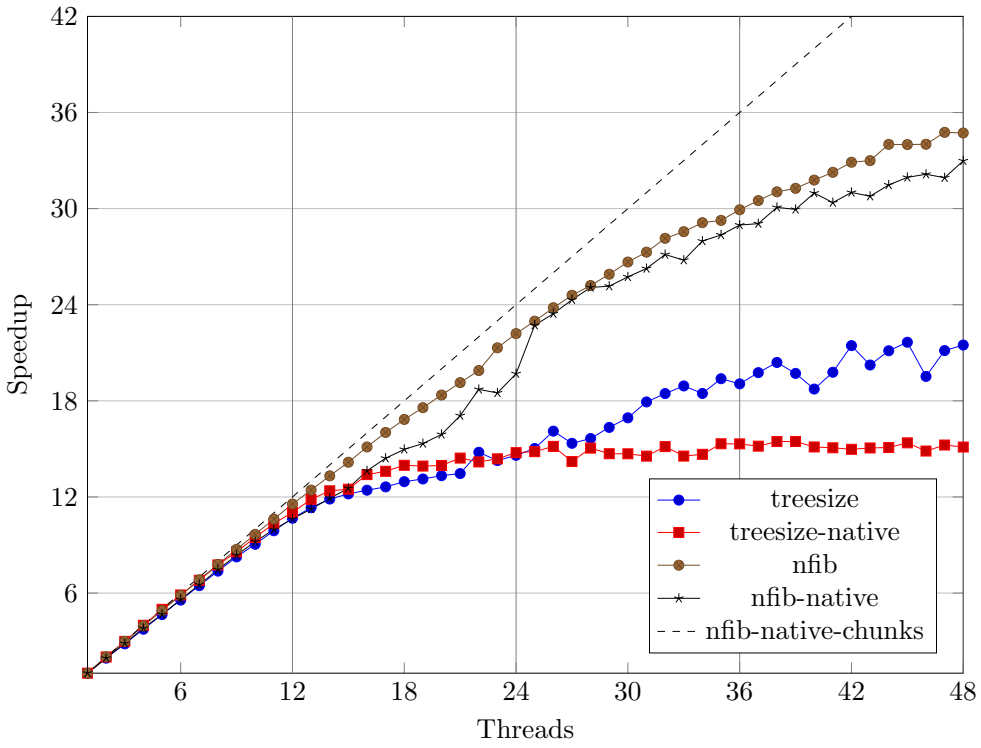


Figure 9.1.: Relative speedup.

Relative Speedup

Figure 9.1 shows the relative speedup that we have measured. The dashed line shows the ideal speedup that can be achieved, assuming a linear speedup as the number of threads increases. The vertical lines show the boundaries of the processor nodes of our testing system.

Using up to twelve threads, for both example programs we managed to get a relative speedup of around 10.6x, and for the non-native `nfib` program, we even got a speedup of 11.5x. These speedups are very good, considering that they are close to the ideal speedup of 12x. We think that the speedup is not ideal due to communication overhead between the reduction threads, and due to speculative evaluation of redexes.

Using up to 48 threads, the relative speedup scales even further for the `nfib` programs, up to 34x for the non-native `nfib` program, and up to 32x for the native nfib program. For the non-native `treesize` program we get a relative speedup of about 21x and for the native version we got a speedup of about 15x. As soon as more than one processor node is used, the increase in speedup suddenly drops for the treesize program. We suspect that this happens because the threads have to access memory on another NUMA node

when scaling beyond one node, which takes longer than accessing memory locally. The `nfib` program does not show this behaviour, and keeps scaling quite well beyond one processor node. We think that this is because the `nfib` program is much less data intensive than the `treesize` program.

Overhead of the Parallel Graph Reducer

	treesize	treesize-native	nfib	nfib-native
Serial	2666 ms	819 ms	3294 ms	626 ms
Parallel	3243 ms	1291 ms	4162 ms	819 ms
Overhead	21.6%	57.6%	26.4%	30.1%

Figure 9.2.: Comparison of serial and parallel graph reduction.

Figure 9.2 shows concrete execution times obtained from our serial graph reducer and our parallel graph reducer, and the overhead of parallel graph reduction compared to serial graph reduction. The overhead of the `treesize-native` program is twice as large as for the other programs. We suspect that this is because the treesize function has to also reduce the tree, which requires additional communication compared to the `nfib` program to share the results of this computation. Furthermore, sharing reduction results represents a larger proportion of the execution time for the `treesize-native` program than for the non-native `treesize` program, which might explain the large difference in overhead between the two.

Discussion

An interesting result is that we could exploit the inherit parallelism of functional programs without using annotations to limit concurrency. However, as discussed in Section 6.2, fully lazy programs can not be parallelised without annotations using this method, as laziness forces programs to be executed sequentially. However, our method depends on laziness to share results early. Using eager evaluation, the sharing nodes are only updated once the eager computation has been completely evaluated. So to use this method, we either have to annotate programs to introduce parallelism explicitly, or there has to be a mix between lazy evaluation and eager evaluation. Furthermore, the ratio between the amount of lazy evaluation and eager evaluation determines the amount of communication overhead and speculative execution. Performing a higher ratio of work eagerly leads to lower communication overhead, but to higher levels of speculation. Vice versa, performing a higher ratio of work lazily leads to higher communication overhead, but to lower levels of speculation.

We have to note that our experiments with this method of graph reduction are limited. The reason for this is that our main focus is on transaction processing, and not the development of a parallel graph reducer. Further investigation with more complex algorithms is needed to further evaluate the parallel graph reducer.

9.3. Transaction Processing - Concurrency

In this section, we discuss experiments to investigate the behaviour of our prototype implementation when executing update transactions concurrently with read transactions. We want to show that there is overlap in the execution of transactions, and we want to measure how this overlap affects execution times of transactions and the memory usage.

Setup

All benchmarks are run on a state that is initialised with a map that contains a mapping from the keys $\{0, \ldots, n\}$ to the value 0, where n depends on the specific benchmark. The map is implemented as a binary search tree. As we do not have an algorithm to balance trees dynamically without blocking access to the map, we use a non-balanced tree for the map implementation. Values are inserted in a random order as to obtain a reasonable on-average balancing of the tree.

For this benchmark we use only a single thread, so we only measure concurrency obtained through lazy reduction of the state. This means that we only measure concurrency between one transaction that reads the state and multiple transactions that update the state. At the start of the benchmark, we update the state to increment all values in the map by one. This new state is constructed lazily, meaning that only those parts of the state are evaluated that are actually read. To perform our benchmarks, we read individual values at random from the map, chosen by the benchmarking program. When we read a value, only that part of the state is reduced that is needed to produce the requested value.

Response Time

In our first benchmark, we want to show that transactions are actually executed concurrently. To do this, we update the state, and we read a randomly chosen value from the map in the updated state and measure the response time. Next, we force the reduction of the new state, and show that the time required for full reduction is much higher than for reading just a single value. Thereby we essentially show that the read transaction must have been performed concurrently with the update transaction.

	update	read	force
1 update	89 μs	97 μs	1405 ms
2 updates	144 μs	149 μs	2390 ms
4 updates	269 μs	125 μs	4323 ms
8 updates	516 μs	255 μs	8214 ms
16 updates	1062 μs	387 μs	16402 ms

Figure 9.3.: Transaction execution times.

Table 9.3 shows the results of our experiments. We executed the benchmark for one up to 16 updates prior to reading the state. The table show that updating is very quick

in all cases, because the new state is not constructed until it is read. Next, we can see that reading a random value from the updated state is very quick as well. This is because only those parts of the updated state are evaluated that are required to fetch the value being read. Finally, we forced the reduction of the new state to show the actual execution time for full reduction of the state. This benchmark essentially shows that the execution of the read transaction was done concurrently with constructing the new state. Showing that reading and updating the state is performed concurrently because of lazy evaluation of the state.

Effect on Throughput

In the second benchmark, we want to measure how an update transaction affects the execution time of concurrent read transactions. As a baseline, we measure the transaction throughput when no updates are going on. We then update the state, and measure the throughput of read transactions

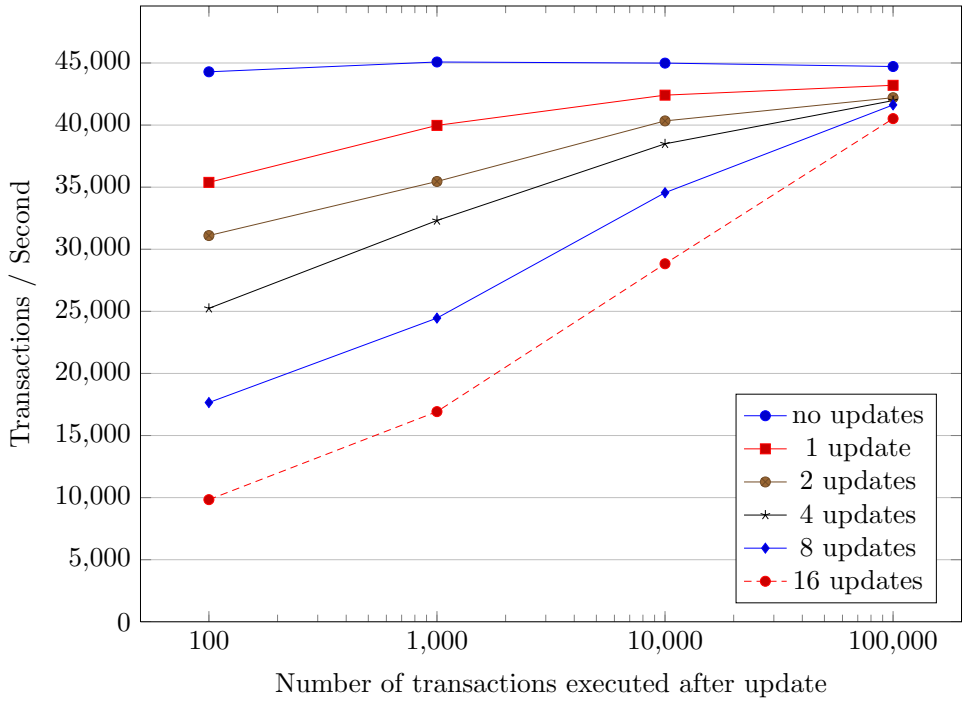


Figure 9.4.: Influence of updates on transaction throughput for a map of size 10,000.

Figure 9.4 shows how the throughput of read transactions is affected by concurrent update transactions. For the first few transactions, quite a lot of work has to be performed to get to a leaf node, where the amount of work depends on the amount of update transactions executed. This means that initially the performance is much lower than the case without updates. As more read transactions are executed, increasingly less

work has to be performed as a larger part of the state has been reduced. When enough read transactions have been executed, performance goes up to nearly the level when no updates where executed.

Effect on Memory Usage

In the final benchmark, we want to measure how an update transaction affects the memory usage when executing concurrent read transactions. Here we essentially do the same as the previous benchmark, but instead we measure the memory usage at certain points during the benchmark.

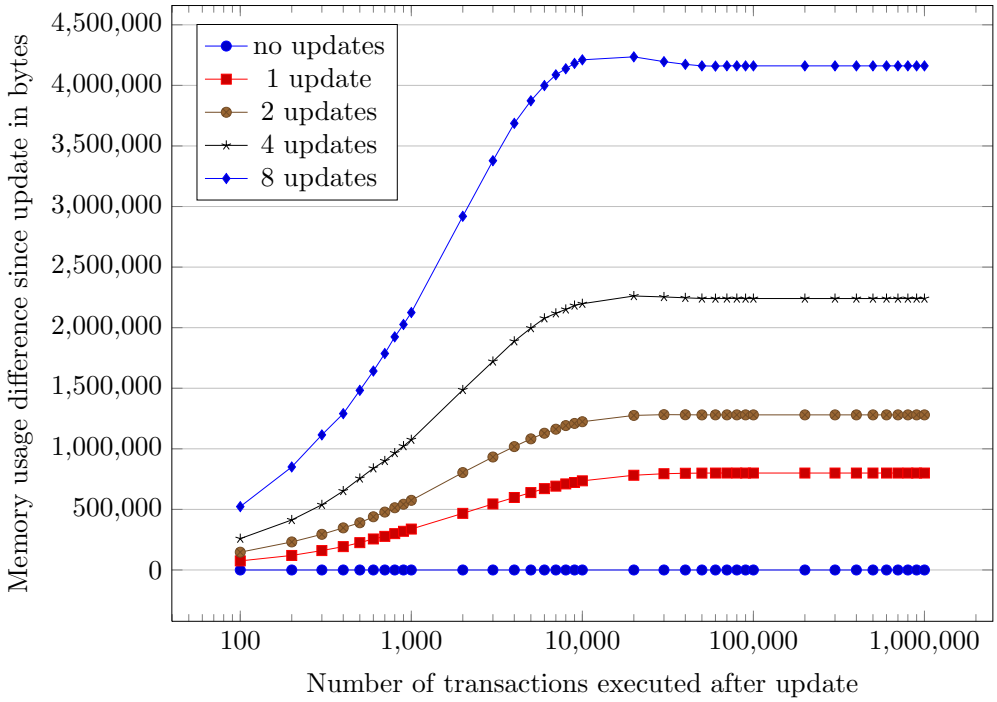


Figure 9.5.: Influence of updates on memory usage for a map of size 10,000.

Figure 9.5 shows the memory usage of our prototype while read transactions are executed on a lazily updated state. Memory usage increases during the execution of a read transaction because redexes are created for all side-branches in the tree while reducing the main branch to get to the value we requested. Intuitively, memory usage should go down at a certain point when the majority of the values have been read, as most redexes should be evaluated by then. We did not observe this behaviour in our prototype, and profiling showed that there are two reasons why this is the case.

One problem is that from the state root, a reference is kept to all sharing nodes, even when they all have been reduced already and are not needed anymore. When the benchmarks starts, there is just one sharing node for the map function at the root of the tree.

After all reads have been processed, this initial sharing node points to its result, which now contains two sharing nodes. This continues recursively, such that almost all data nodes have a sharing node.

Another problem, which accounts for the majority of the increase in memory usage, is due to our map data structure implementation. The map function is pushed into the leaf nodes of the map data structure, but the leaf nodes never have to be evaluated for any read. This means that when each value has been requested in the map, each terminating tree branch contains a sequence of update redexes applied to its leaf node.

This last problem shows a general problem with the theoretical model of using lazy reduction of state for concurrency, because even in a theoretical setting such space leaks will show up. A general solution to solve both problems is to force the evaluation of update transactions. This will clean up the sharing nodes, as well as forcing the reduction of the redexes applied to the leaf nodes. In Section 7.4 we discuss this solution in more detail.

9.4. Transaction Processing - Throughput

In our final set of experiments, we measure how transaction throughput scales with increasing load on the system. First, we discuss how our experiments are set up. Then, we investigate throughput for read transactions. Next, we investigate throughput for update transactions, and finally we investigate how our prototype behaves for a mix of read and update transactions.

Setup

We use essentially the same setup for the state as for the concurrency experiments, where we map the values $\{0, \ldots, 100,000\}$ to the value 0. For these experiments, we issue two types of transactions to the system: update transactions that update a single value, and read transactions that read a single value. As to not overload the system with lazily evaluated updates to the state, an update transaction also read the value it has written, forcing the evaluation of all redexes.

To simulate increasing load on the system, we use multiple threads to issue transactions. We use more threads than the available cores in our benchmarking system, as in practice, load may also increase beyond what a machine is able to handle. To create a mix of read and write transactions, each thread chooses whether to issue a read or write transaction at random, weighted by the ratio of reads and writes we want to measure. The graphs in this section only show the total transaction throughput, to obtain the transaction throughput of only reads or only writes, we can simply multiply the total throughput by the ratio.

As we now use multiple threads, the way we sequence transactions will now affect performance. In the benchmarks we measure the transaction throughput for both a readers / writers lock approach to sequencing, as well as the lockless readers approach, as discussed in Section 7.3.

Only Read Transactions

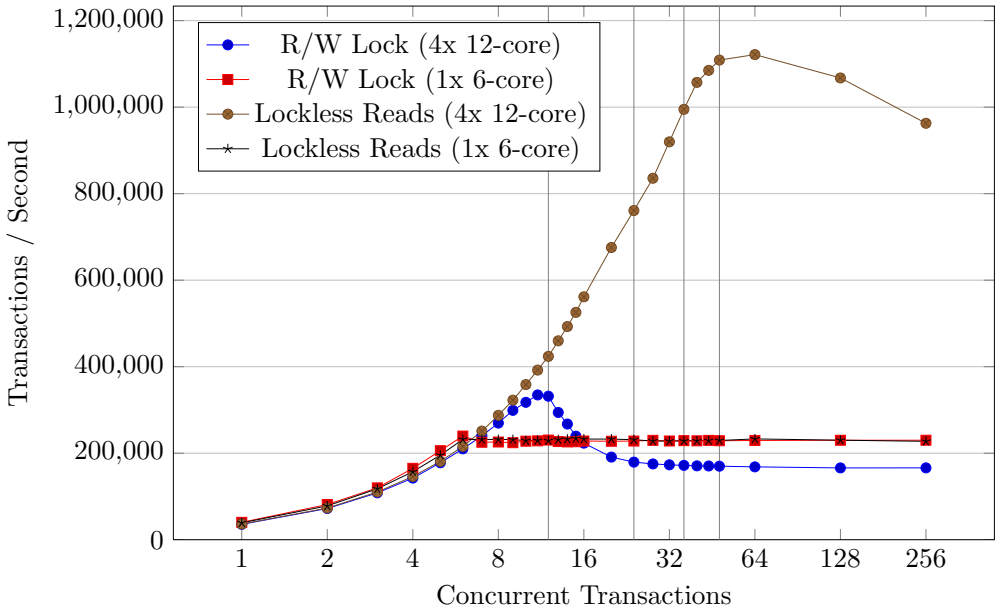


Figure 9.6.: Transaction throughput of reads.

First, we investigate the throughput of read transactions. Figure 9.6 shows the throughput that we have measured. We see that the lockless approach scales very well, from 36,262 transactions per second when there is no concurrency, up to 1,121,480 transactions per second with 64 concurrent transactions per second, a relative increase in throughput of 30.9×. With 256 concurrent transactions, throughput drops to 962,694, probably due to overhead of scheduling the threads.

The locking approach scales much worse, from 35,588 transactions per second using one thread, to 331,801 transactions per seconds with twelve threads, and back to 179,441 transactions per second using 24 threads, after which throughput remains stable. We found that the bottleneck here is locking, because as soon as multiple processor nodes start contending for the lock, there is a lot of communication between the processors. A NUMA aware lock [34] might solve this problem, but we could not test this, as we did not have an implementation of this available in Java.

We have validated that locking is the cause of the bottleneck by running the benchmark using only a single NUMA node of six cores. This can be seen in the graph as the measurements labelled with (1x 6-core). Now we see that there is no slowdown after a peak performance of 230,421 transactions per second is reached using the locking. We also see that the sustained throughput of the locking approach with six cores is 28% higher than when using all 48 cores. Additionally, we see that the locking approach performs about equal to the lockless approach when using six cores.

Only Update Transactions

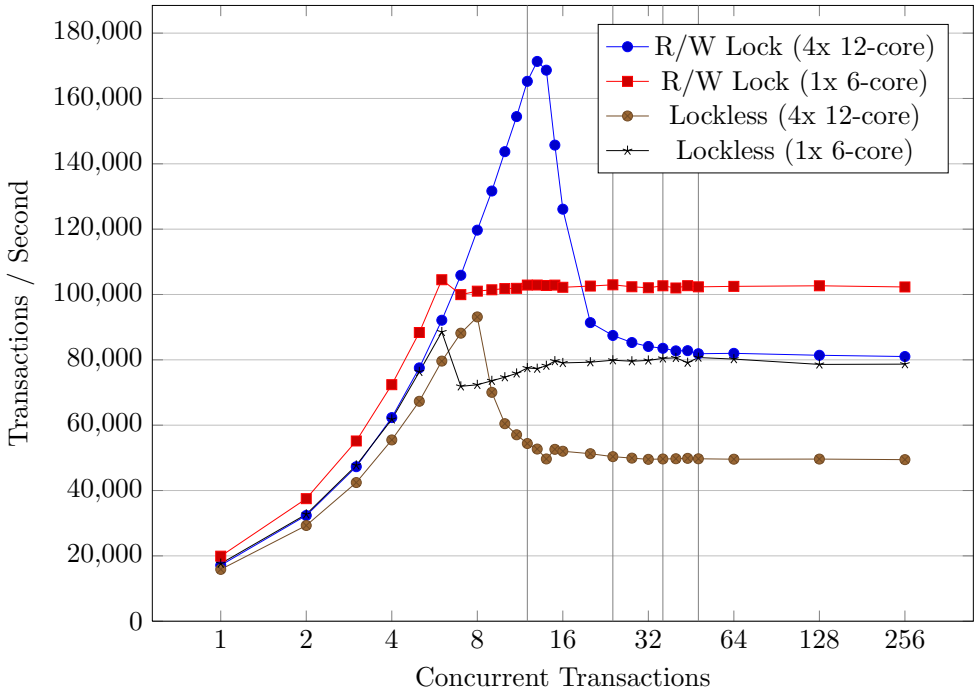


Figure 9.7.: Transaction throughput of updates.

Now we investigate the throughput when only executing update transactions. Figure 9.7 shows the throughput that we have measured. We again see that the locking mechanisms in both approaches suffer from communication overhead as discussed in the previous benchmark. The locking approach peaks at 171.337 transactions per second at 13 concurrent requests, while providing a sustained throughput of around 84.000 transactions per second starting around 20 concurrent requests.

The lockless reads approach peaks at a throughput of 93.139 transactions per second at eight concurrent requests, and maintains a sustained throughput of around 49.000 transactions per second from about twelve concurrent requests. The reason that the throughput of the lockless reads approach is much lower than the locking approach is because the lockless reads approach copies the state bindings for every update, as to provide atomic updates for reads, while the locking updates the bindings in-place.

When using only six cores, we see that the peak transaction throughput is about equal to the sustained throughput. This is about 102.000 transactions per seconds for the locking approach, and about 80.000 transactions per seconds for the lockless reads approach.

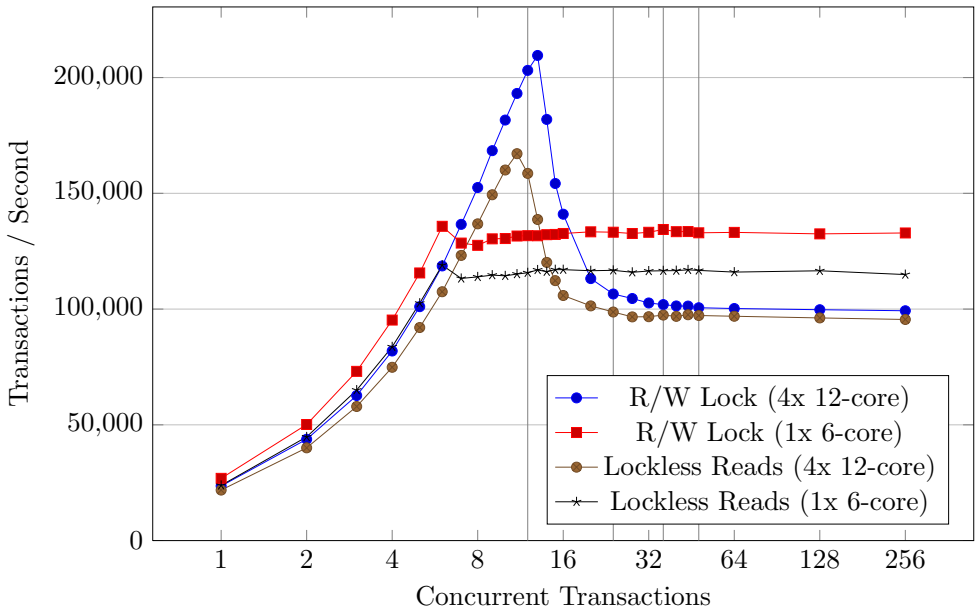


Figure 9.8.: Transaction throughput for a mix of 50% reads and 50% updates.

Mixed Reads and Update Transactions

Finally, we investigate the throughput when executing a mix of read and update transactions. Figure 9.8 shows the throughput for a mix of 50% read transactions and 50% update transactions. What we see is essentially not much different from the experiment where we only perform update transactions, because the transactions are still dominated by locking. However, the lockless reads approach now performs significantly better when using all 48 cores, compared to the results of the update only experiment, as reads are not affected by the locking bottleneck.

Figure 9.9 shows the throughput for a mix of 90% read transactions and 10% update transactions. Now we see that the lockless reads method scales quite well, similar to the results in the read only experiment. However we do still see a drop in performance as concurrency increases beyond 44 concurrent transactions.

Discussion

In general, we found that using a single NUMA node, throughput scales very well. However, using multiple NUMA nodes, locking proves to be a major bottleneck. One approach to solve this would be to use NUMA aware locking techniques [34]. Another option is to investigate a full lockless approach, where binding is performed lazily, and is forced by reading the state. A more advanced approach could be to make the system NUMA aware, distributing the database over multiple NUMA nodes, and migrating reduction threads as much as possible to the NUMA node where the data is.

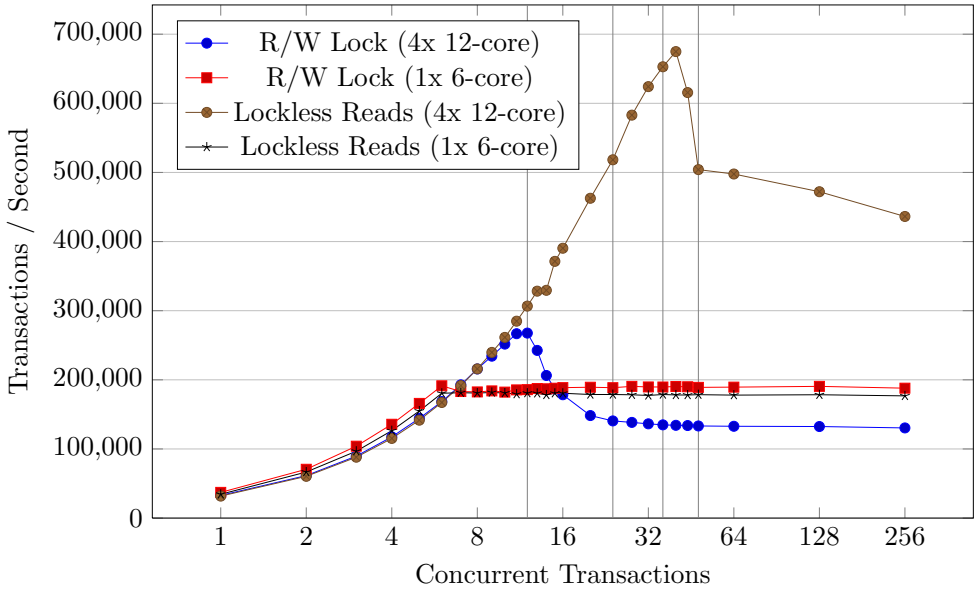


Figure 9.9.: Transaction throughput for a mix of 90% reads and 10% updates.

9.5. Conclusions

In this chapter we have discussed the results of our experiments using our prototype implementation. In summary, we have found the following results:

- Our parallel graph reducer can exploit implicit parallelism in two simple programs, without the need of annotations.
- Transactions that read the state can be executed concurrently with transaction that update the state, resulting in very low response times after large updates have been applied.
- Large updates affect the throughput performance of subsequent reads, but the performance recovers as parts of the state have been reduced.
- Reducing states lazily leads to memory leaks, which shows a problem with the theoretical model in practice, we have also provided a theoretical solution to this problem.
- We have found that transaction throughput scales well with an increasing number of concurrent transactions on a single NUMA node, however when using multiple NUMA nodes locking is a bottleneck due to high contention.

There is still work to be done on resolving the problem of memory leaks and high contention for the lock, but in general we can conclude that our prototype implementation shows promising results.

10. Related Work

Aside from the work by Trinder [38] and Nikhil [28] on which we build, other work has been done that is closely related to our work. In this chapter we first discuss related work in imperative persistent languages. Next, we discuss related work on functional persistent languages. Finally, we discuss related work in graph reduction using randomisation and result sharing.

10.1. Imperative Persistent Languages

The first efforts to integrate transparent persistence with programming languages is the language PS-Algol [4]. More recent efforts to integrate programming with persistence include Persistent Haskell [14] and Persistent Java [25]. Each of these approaches extends an existing programming language with support for automatic persistence of state. In these approaches, objects can be marked as persistent roots, and the system automatically ensures that any object reachable from these roots is stored in persistent memory.

A benefit of these approaches is that they re-use an existing programming language, which makes them more flexible in communication to the outside world than our approach as there is no pre-defined interface. However, these approaches do not provide an interactive interface to execute ad-hoc transactions. The absence of ad-hoc transaction support has been a problem for the evolution of programs implemented in these languages, as it is difficult to change the structure of existing data. Another difference from our model is that these approaches only provide transaction support at the data level, so they are susceptible to problems such as blocking and starvation.

Another approach is Prevayler for Java [2], which is a library that provides mechanisms for transparent persistence and transaction support to Java. In this approach, the programmer implements a fixed set of transactions. Transactions are executed sequentially to avoid concurrency conflicts. Durability is guaranteed by journaling transactions and checkpoints of the state are created by serialising the state. The main strength of this approaches is the simplicity of implementation. However, it suffers from many of problems as the persistent outlined above. Additionally, the state is limited to main-memory, and it does not support concurrent execution of transactions.

Yet another approach are object databases [17], which avoid the object-relational impedance mismatch by using the object model as the data model for the database. Object

databases usually support the object query language to perform ad-hoc queries. Additionally, an object-oriented programming language can be built into an object-database, making object databases similar to persistent languages. To manage concurrent execution of transactions in object-databases, usually concurrency control techniques from databases are used. However, this means that transactions can still be aborted due to concurrency control issues, providing the possibility of starvation when performing large updates. In contrast, our approach derives concurrency from the parallel nature of functional languages, having the benefit that transactions are never aborted, allowing a higher level of concurrency than traditional concurrency control methods, and allowing transactions to be executed in parallel.

10.2. Functional Persistent Languages

In 1991, Nikhil and Heytens [29] described the implementation of the persistent object system AGNA, which is based on a functional language. The main focus in his implementation is to exploit the parallel nature of functional programs for the parallel execution of transactions, and the implementation of storage of states in persistent memory. While they implemented parallel execution of individual transactions, they did not implement the concurrent execution of multiple transactions simultaneously. Furthermore, their persistence method does allow storage of states larger than main memory, which is implemented by paging memory from persistent storage in main memory. However, their implementation does not address durability, and they do not consider snapshotting of ongoing computation.

In 1993, Akerholt et al. investigated the performance of their parallel graph reducer GRIP [3] for functional transaction processing. While the main focus of their work is GRIP, they implemented the functional transaction processing model by Trinder [38], which supports concurrent execution of transactions. However, in their experiments they only evaluated the parallel speedup of their graph reducer, and they did not investigate the behaviour of transactions when executing concurrency. Further, they did not address persistence or interaction models.

In 1993, McNally [26] investigated methods for the interaction with functional persistent languages, as well as creating a practical implementation of a persistent functional language called STAPLE. McNally developed two interaction models, a stream persistence model that allows the interactive evaluation of expressions in an environment, and a module based persistence model to load pre-defined modules into the system. McNally uses an of-the-shelf persistent object store [9] to implement transparent persistence. The STAPLE system uses the PCASE machine developed by McNally, which operates directly on the persistent object store, thereby providing transparent persistence of data, as well as persistence of ongoing computation.

Stored transactions in our language are similar in functionality to the module based persistence approach by McNally, as they support the definition of pre-defined transactions. However, STAPLE only supports the addition and removal of bindings in the module

based persistence model, and during interactive sessions bindings can not be created or removed. Our prototype implementation does support adding and removing of bindings, which is implemented by resolving references to templates statically. Our approach to persistence differs from the approach chosen b McNally in that we do not use a persistent object store to implement persistence. Instead, we have investigated methods that are optimised for functional languages. Furthermore, a limitation of STAPLE is that it does not support concurrent execution of transactions.

A more recent project is that integrates functional languages with transactions and persistent storage is ACID State for Haskell [1]. It is similar to Prevayler as discussed in the previous section, however is does support concurrent execution of transactions by forking a reduction thread for every update to the state. We will further study their approach to see how concurrent execution of transactions differs from our approach.

10.3. Parallel Graph Reduction

The idea of randomisation and result sharing for load balancing of work in parallel systems has already been applied successfully in the context of model checking for the parallel exploration of state-spaces [16]. The study of results in this field may lead to interesting insights for parallel graph reduction.

An approach similar to ours been studied for the parallelisation of operations on binary decision diagrams in the context of symbolic model checking by van Dijk [15]. Different from our approach, they have implement result sharing through a lockless memoisation cache, and they use a fast random number generator to implement randomisation of threads. Furthermore, they have compared the work-stealing approach, implemented using Cilk [8], to randomisation and result sharing, and found that randomisation and result sharing performs about equal to the work-stealing approach.

11. Conclusions

In this thesis we explore the development of a persistent functional language for concurrent transaction processing. In this chapter, we first discuss how our contributions achieve the goals that we set out in the introduction. Finally, we discuss directions for further research.

11.1. Goals and Contributions

In the introduction we have described the goals that we want to achieve in this thesis. In this section we describe how our contributions have achieved these goals.

Development of a Transactional Functional Language

The first goal of this thesis is the development of a functional language for transaction processing.

In Chapter 4 we have developed of a language that can be used to describe functional transactions. A transaction in this language describes updates to the state, and may contain a result expression that is evaluated in the context of the current state. The key feature of our language is that we introduce two kinds of variables, one that refers to the current state, and one that refers to the next state. Furthermore, we introduced stored transactions, which can be invoked through an external interface, providing a basic building block for the implementation of transaction processing applications.

In Chapter 5 we have developed a graph reducer that serves as the basis for the implementation of our language. The main feature of our graph reducer is that it allows the dynamic creation of functions. Instead of maintaining a global environment of function templates as is done in template instantiation, we statically resolve references to function templates. This makes function templates anonymous such that they can be garbage collected automatically.

Finally, in Chapter 7 we have developed a transaction manager for our language model. The transaction manager allows the execution of a stream of transactions on a state, producing a stream of results.

Concurrent Execution of Transactions

Our second goal is to allow concurrent execution of transactions. Concurrency is already partly supported through lazy reduction of states, but we also need to support parallel reduction of states, and correctly handle concurrent updates of the system state.

To support parallel reduction of states, in Chapter 6 we have developed a new method for parallel graph reduction. Our method is based on correctly sharing reduction results between threads to allow concurrent reduction, and randomising the reduction order of strict function arguments to perform load balancing to allow parallel reduction. In Chapter 9 we show that our method is able to obtain a nearly ideal relative speedup on a single multi-core processor for some simple programs. Interestingly, we did not have to annotate our program with parallelisation strategies to achieve a decent speedup, in contrast to work-stealing approaches to load-distribution.

In Chapter 7 we have developed two methods to handle concurrent updates to the state. One method is based on locking the state for every operation, and allowing operations to update the state in-place. The other method is based on updating states atomically, allowing reads to be lockless. In our experiments in Chapter 9 we found that for a high update load, updating in-place performs better, while for high read loads there is little difference between both methods. A limitation of our experiments is that our locking implementation is not NUMA aware, so we could not measure the performance of both methods accurately on a NUMA system.

In Chapter 9 we have also performed experiments with concurrency solely through lazy evaluation of states. We found that space leaks may build up in the state due to lazy evaluation. Also, we found that lazy evaluation may lead to long chains of redexes in the state, which caused stack overflow errors in our prototype implementation. We have developed a theoretical solution to this problem, where we force the evaluation of transactions and limit the number of concurrent transactions in progress.

Efficient Persistence

Our third and final goal is to efficiently store functional states in persistent memory.

In Chapter 8 we investigated three methods to store functional states in persistent memory: snapshotting, log-structured storage and a combined approach. Each method supports durability of transactions that update the state through journaling. Snapshotting allows the creation of checkpoints of ongoing computations, but it only supports states that fit in main-memory, and it suffers from long recovery times. Log-structured storage allows states that are larger than main-memory, and has nearly instant recovery times, but it does not support checkpointing of ongoing computations. We combined both approaches where we use snapshotting for the unevaluated part of the state, and we use log-structured storage for the evaluated part of the state. The combined approach allows states larger than main-memory, snapshotting of ongoing-computations, as well as quick recovery times.

Finally, as a proof of concept that our language can indeed support persistence, we have implemented journaling together with a simple variant of the snapshotting approach that does not allow snapshotting of computations.

11.2. Limitations

While we have developed a persistent functional language, the main limitation of our work is that we have not explored the use of this language in practice. That is, we have not developed an actual DBMS, or implemented a realistic transaction processing system in our language. However, this thesis is only the first step of a much larger project where we plan to investigate these topics by means of a case study.

A limitation of our approach to persistent functional languages is that we only consider shared memory systems. Compared to the traditional approach, we can not integrate data from multiple DBMS's in an application, and we can not perform transactions between multiple systems. Additionally, our system is a single point of failure in our current architecture, and we can not distribute workload or data storage among multiple systems. We hope to address these issues in future investigations.

11.3. Future Work

In this section we present ten directions for further research towards the goal of using functional language for transaction processing.

Implementation: We will investigate the implementation of forcing the evaluation of transactions as discussed in Chapter 7. We will also investigate the implementation of our methods to store states in persistent memory, as discussed in Chapter 8.

Functional Databases: In this thesis we have not discussed the application of our language for the construction of functional databases. We will investigate the implementation of data models and common database features in our prototype language. Furthermore, we also want to investigate the possibilities of adding DBMS features such as role based security mechanisms and automatic consistency checks.

Handling Runtime Problems: In practice we encounter runtime problems such as running out of memory, stack overflows, and non-termination. For example, if we execute a transaction that does not terminate while constructing a new state, we need a method to undo the execution of this transaction. Further research is needed to handle such situations.

Type Checking: We will investigate how types can be defined and checked dynamically in our language model. One of the challenges here is that type checking must be performed as part of a transaction, as types may change between type checking a transaction and executing a transaction. Additionally, methods need to be developed to allow data types to be redefined, to allow evolution the of database schemas.

Concurrent Data Structures: Algorithms and data structures that provide concurrency between transactions under lazy evaluation have not yet been investigated in existing literature. In particular, a concurrent functional balanced search tree is

required for the implementation of functional databases. Expressions can be written in multiple ways, with the same result, but with different behaviour regarding concurrency between transactions. We will formalize the notion of concurrency of functional transactions, and investigate the automatic optimisation of expressions to remove concurrency bottlenecks.

Optimistic Concurrency Control: Concurrency obtained through lazy evaluation will not be enough for all situations. Sometimes transactions in this model will simply block access to the database due to functional dependencies. To resolve this problem we will investigate the possibilities of introducing optimistic concurrency control for updates. One idea is to split a transaction into multiple transactions that are each non-blocking, but which preserve the ACID properties as a whole. We will investigate the correctness and composition of such transactions, as well as the possibility of automatically splitting transactions to increase concurrency. Additionally, this model may also provide a basis for interactive transactions.

Reducing Memory Usage: As a result of the experiments that we have performed on our prototype, we found that the system uses quite a lot of memory. This is because the data structure consists mostly of pointers, which produce a lot of overhead. To resolve this problem, further research could look into data structures that are more efficient regarding memory usage. An idea is to have trees with wider branches, as this reduces the amount of pointers in the data structure. To conveniently implement this, we think that an array data type would be convenient in the language.

Parallel Graph Reduction: We have described a method to perform graph reduction by sharing results between reduction threads, and randomisation of their reduction order. However, our experiments with this method of graph reduction are of limited scope, as it is not the main topic of this thesis. We have some ideas for further research:

- It would be interesting to know how sharing results and randomisation compares to the work-stealing approach. Additionally, it might be interesting to investigate if work-stealing can be combined with result sharing and randomisation. Also, further investigation of methods for the randomisation of the reduction order of threads is needed.
- Communication overhead between threads can be reduced by increasing the granularity of tasks. An approach to do this is to mix lazy evaluation and eager evaluation, as discussed in Section 9.2. It would be interesting to investigate this further, and develop methods to do this automatically.

Scheduling: In our current implementation we use the operating system scheduler to schedule the execution of concurrent transactions by assigning a fixed number of graph reduction threads to each transaction. We will investigate scheduling of reduction on a fixed number of operating system threads, as to reduce the

overhead of spawning threads. Additionally, sometimes the evaluation of updates may become irrelevant due to newer updates. In our current model these reduction of these updates will continue until normal form is reached. It would be more efficient if reduction that is not relevant anymore could be stopped. An idea is to remove these from the schedule during garbage collection.

Distributed Persistent Languages: Another direction for future research is to investigate the possibilities of distributing persistent functional languages over multiple machines. One approach is to replicate the state among multiple computer systems for increased performance of read-only transactions and for fault tolerance. Further research can also look into methods for distribution of update transactions, or sharding the state among multiple systems. Finally, it would be interesting to investigate a model based on eventual consistency, which could lead to highly scalable persistent functional languages.

Bibliography

[1] ACID State manual: http://happstack.com/docs/crashcourse/acidstate.html, August 2012.

[2] Prevayler website: http://prevayler.org/, August 2012.

[3] Akerholt, G., Hammond, K., Jones, S. P., and Trinder, P. Processing transactions on grip, a parallel graph reducer. In *PARLE93, Parallel Architectures and Languages Europe, number 694 in Lecture Notes in Computer Science* (1993), Springer, pp. 634–647.

[4] Atkinson, M., Bailey, P., Chisholm, K., Cockshott, W., and Morrison, R. PS-algol: A Language for Persistent Programming. In *10th Australian National Computer Conference, Melbourne, Australia* (1983), pp. 70–79.

[5] Barendregt, H. P., and Barendsen, E. Introduction to Lambda Calculus (Revised Edition), 2000.

[6] Ben-Ari, M. *Principles of concurrent and distributed programming.* Prentice-Hall, Inc., 1990.

[7] Blackburn, S., and Zigman, J. N. Concurrency - the fly in the ointment? In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems* (1999), Morgan Kaufmann Publishers Inc., pp. 250–258.

[8] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming* (1995), PPOPP '95, ACM, pp. 207–216.

[9] Brown, A. *Persistent Object Stores.* PhD thesis, University of St Andrews, 1988.

[10] Bruijn, N. G. D. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae 34* (1972), 381–392.

[11] Brus, T. H., van Eekelen, C. J. D., van Leer, M. O., and Plasmeijer, M. J. Clean: A language for functional graph rewriting. In *Proceedings of Functional programming languages and computer architecture* (1987), Springer-Verlag, pp. 364–384.

[12] Church, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 2 (April 1936), 345–363.

[13] Codd, E. F. A relational model of data for large shared data banks. *Commun. ACM 26*, 1 (Jan. 1983), 64–69.

[14] Davie, T., Hammond, K., and Quintela, J. Efficient persistent Haskell. In *Draft proceedings of the 10th workshop on the implementation of Functional Languages* (1998), University College London, pp. 183–194.

[15] Dijk, T. The parallelization of binary decision diagram operations for model checking. Master's thesis, University of Twente, 2012.

[16] Dwyer, M. B., Elbaum, S., Person, S., and Purandare, R. Parallel randomized state-space search. In *Proceedings of the 29th international conference on Software Engineering* (2007), ICSE '07, IEEE Computer Society, pp. 3–12.

[17] Garcia-Molina, H., Ullman, J. D., and Widom, J. *Database Systems: The Complete Book*, 2 ed. Prentice Hall Press, 2008.

[18] Gray, J. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7* (1981), IEEE Computer Society, pp. 144–154.

[19] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[20] Harris, T., and Singh, S. Feedback directed implicit parallelism. *SIGPLAN Not. 42*, 9 (Oct. 2007), 251–264.

[21] Hearn, A. C. Standard lisp (reprint). *SIGSAM Bull.*, 13 (Dec. 1969), 28–49.

[22] Hulse, D., and Dearle, A. A log-structured persistent store. In *Proceedings of the 19th Australasian Computer Science Conference* (1996), pp. 563–572.

[23] Kifer, M., Bernstein, A., and Lewis, P. M. *Database Systems: An Application Oriented Approach, Complete Version (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[24] Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K., and Trinder, P. Seq no more: better strategies for parallel haskell. In *Proceedings of the third ACM Haskell symposium on Haskell* (New York, NY, USA, 2010), Haskell '10, ACM, pp. 91–102.

[25] Marquez, A., Blackburn, S., Mercer, G., and Zigman, J. Implementing orthogonally persistent java. In *Revised Papers from the 9th International Workshop on Persistent Object Systems* (2001), Springer-Verlag, pp. 247–261.

[26] McNally, D. *Models for Persistence in Lazy Functional Programming Systems*. PhD thesis, University of St Andrews, 1993.

[27] Netzer, R. H. B., and Miller, B. P. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems 1*, 1 (Mar. 1992), 74–88.

[28] Nikhil, R. Functional databases, functional languages. In *Data Types and Persistence, Proceedings of the First Workshop on Persistent Objects* (1985), Springer-Verlag, pp. 299–313.

[29] Nikhil, R. S., and Heytens, M. L. Exploiting parallelism in the implementation of Agna, a persistent programming system. In *Proceedings of the Seventh International Conference on Data Engineering* (1991), IEEE Computer Society, pp. 660–669.

[30] O'Sullivan, B., Goerzen, J., and Stewart, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[31] Peyton Jones, S., Ed. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[32] Peyton Jones, S., and Lester, D. *Implementing functional languages: a tutorial*. Prentice Hall, 1992.

[33] Peyton Jones, S. L. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

[34] Radovic, Z., and Hagersten, E. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture* (2003), HPCA '03, IEEE Computer Society, pp. 241–.

[35] Rainey, M. A. *Effective scheduling techniques for high-level parallel programming languages*. PhD thesis, University of Chicago, 2010.

[36] Su, Z., and Wassermann, G. The essence of command injection attacks in web applications. In *Symposium on Principles of Programming Languages* (2006), ACM, pp. 372–382.

[37] Tremblay, G., and Gao, G. R. The impact of laziness on parallelism and the limits of strictness analysis. In *Proceedings of high performance functional computing* (1995), pp. 119–133.

[38] Trinder, P. *A Functional Database*. PhD thesis, University of Oxford, 1989.

[39] Turner, D. A. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of Functional programming languages and computer architecture* (1985), Springer-Verlag New York, Inc., pp. 1–16.