Introducing Parallelism in an Educational Mathematics System Developed in a Functional Programming Language

Mathias Lehnfeld

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2014

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 18, 2014

Mathias Lehnfeld

Contents

Acknowledgements

This page is dedicated to all those who were in any way involved with the outcome of this thesis.

First of all I want to express my utter gratitude towards my parents. They were there for me, constantly encouraging and supporting me generously in so many ways. Then there are my sisters Clara, Vroni and Anita, whose wonderful personalities make me a happier person. I want to thank my relatives and friends for being such an important part of my life and of who I am. Also, my colleagues are funny and inspiring people without exception and they have made my time as a student a pleasure.

Last but most certainly not least I am exceedingly grateful to my supervisors, Mag. Volker Christian and Dr. Walther Neuper.

Walther Neuper's enthusiasm for his project and his eager cooperation and supervision were extremely motivating and made the whole process of creating this thesis very efficient. His gentle and wise personality and his trust in me were very encouraging.

I owe my great appreciation to Volker Christian for his friendly personal and competent, professional supervision and his interest in my topic. I am overly thankful for his trust, support and encouragement to pursue my interests.

For Franz, my uncle and godfather.

Abstract

Multi-core hardware has become a standard in personal computing. Similar to the *software crisis* around 1970, writing software which is able to exploit the new, parallel hardware is still a difficult, error-prone process.

Due to a side effect free programming style, functional programming languages are said to be specifically suitable for this kind of tasks. One application of such languages are theorem provers. Their foundation in mathematics and logics makes functional programming the paradigm of choice. *Isabelle* is an interactive theorem proving framework whose back-end has been implemented in *Standard ML*. Development efforts during recent years have enabled the system to efficiently use the full processing power available on current multi-core architectures. The educational mathematics system *Isac* draws on *Isabelle*'s functionality and was thus until recently the only project of its kind to be based on theorem proving technology.

This thesis documents the introduction of *Isabelle*'s parallelism concepts to *Isac* and outlines related technologies such as parallelism fundamentals, various approaches to parallelism in functional programming and computer theorem proving. Furthermore, the architectures and concepts of *Isabelle* and *Isac* are discussed, both including means of communicating with the *Java Virtual Machine* for their front-ends. Thereby the strengths of functional and imperative programming are combined and both paradigms are utilized to solve the respective problems for which they are particularly appropriate. The results of the parallelization are promising and will enable a high number of students to use one single, responsive *Isac* server for calculations simultaneously.

Kurzfassung

Multicore-Prozessoren sind im Personal Computing Bereich mittlerweile ein etablierter Standard. Ähnlich der *Softwarekrise* um 1970 ist aber die Entwicklung von Software, die diese Hardware effizient nutzen kann, immer noch ein schwieriger, fehleranfälliger Prozess.

Funktionale Programmiersprachen basieren auf einem Programmierstil ohne Nebeneffekten und sollen daher für diese Art von Aufgaben besonders geeignet sein. Eine Anwendung solcher Sprachen sind computerbasierte Theorembeweiser. Wegen ihrer Begründung in Mathematik und Logik liegt funktionale Programmierung als Paradigma besonders nahe. *Isabelle* ist ein Framework für interaktives Beweisen von Theoremen, dessen Backend in *Standard ML* entwickelt wurde. Die Implementierungsarbeiten der letzten Jahre haben dazu geführt, dass das System nun die Ressourcen moderner Multicore-Prozessorarchitekturen besonders wirkungsvoll einsetzen kann. Das Mathematiklernsystem *Isac* baut auf *Isabelle* auf und war damit bis vor Kurzem das einzige seiner Art, welches auf Theorembeweisertechnologien basiert.

Diese Arbeit dokumentiert die Einführung von *Isabelle*s Parallelitätskonzepten in *Isac* und gibt einen Überblick über dafür relevante Technologien wie Grundlagen der Parallelität, unterschiedliche Ansätze zu Parallelität in funktionaler Programmierung und maschinengestütztes Beweisen. Außerdem werden die Architekturen *Isabelle*s und *Isac*s vorgestellt, die beide Mechanismen zur Kommunikation mit der *Java Virtual Machine* für ihre Frontends beinhalten. Damit werden die Stärken funktionaler und imperativer Programmierung kombiniert und beide Paradigmen jeweils zur Lösung jener Probleme eingesetzt, für die sie besonders angemessen sind. Die Ergebnisse der Parallelisierung sind vielversprechend und werden es ermöglichen, dass eine große Anzahl von Schülern und Studenten gleichzeitig einen schnell reagierenden *Isac* Server für Berechnungen beanspruchen kann.

Chapter 1

Introduction

1.1 Motivation

Moore's Law [65] states that the density of transistors which can be placed on a single chip doubles about every two years. Clock speeds hit an upper bound around the year 2004 (see fig. 1.1) due to thermal issues [84] and transistor sizes cannot be reduced any more [28]. Therefore most CPUs now come with multiple processing cores. This poses a challenge to the software running on these CPUs because it must support parallel or concurrent execution in order to efficiently exploit a computer's full processing power. Concurrency allows for multitasking and asynchronous computations can improve responsiveness even on uniprocessors (see section 2.4). But writing efficiently parallel or concurrent software is hard [85]. Now that multi-core architectures are commonplace and highly parallel GPUs are beginning to get exploited for general-purpose computing [68], concurrent processes and threads can effectively be running simultaneously and therefore significantly faster.

Functional programming is a declarative paradigm based on the lambda calculus. It has been claimed that it is specifically suited not only for prototyping but also for efficient parallelism due to some of its defining features such as immutable data, statelessness and the lack of side effects (e.g. [17]).

The interactive theorem prover *Isabelle* [69] is a comprehensive, international software project. Its internal logic has been written entirely in *Standard ML* which later was embedded in a *Scala* environment to simplify interaction with the underlying operating system and software built on top of *Isabelle*. As the use of multi-core systems became more common, *Isabelle*'s computationally complex logic was required to exploit the newly available processing power. Makarius Wenzel, one of the leading developers of *Isabelle*, parallelized essential parts of the system within a few man months (scattered over about two years). This is a surprisingly little amount of effort considering the project's size and complexity. A significant amount of the

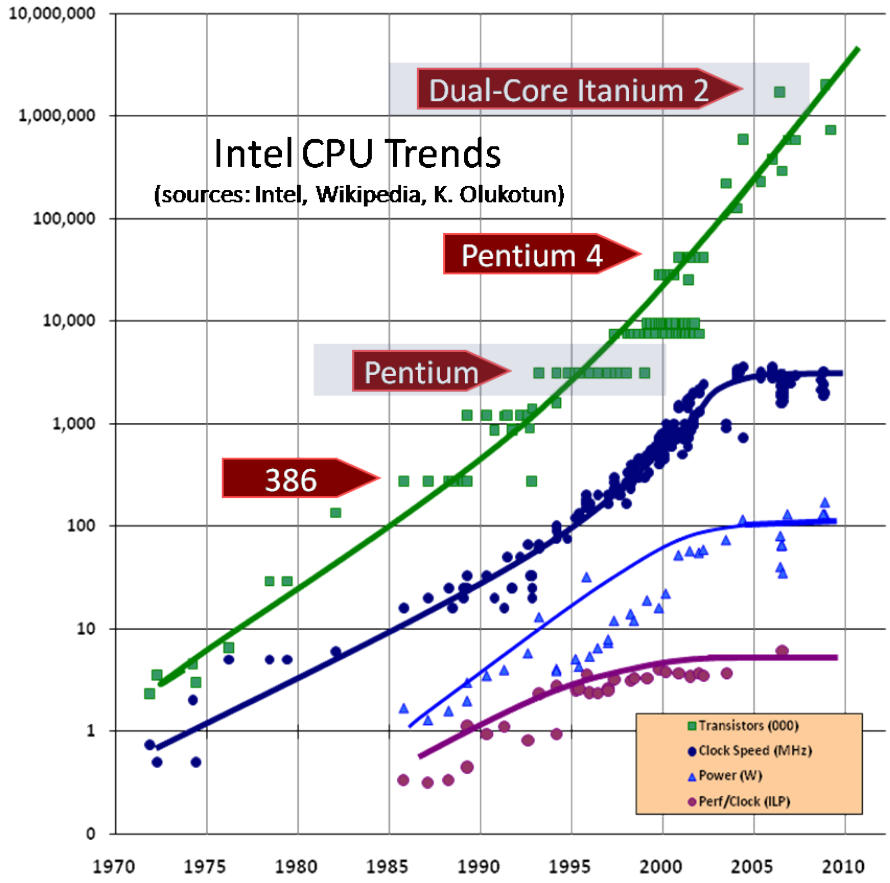

Figure 1.1: Intel CPU trends (source: [84]).

work was required due to the use of impure code, facilitated by *Standard ML* and *Scala*, i.e. work concerning purification in terms of the functional paradigm. *Isabelle/Scala* provides tools for interaction between front-end and proof engine and contains object-oriented components to deal with inherently stateful aspects of the development environment *Isabelle/jEdit*.

The educational mathematics system *Isac* is being prototyped at *Graz University of Technology*. The prototype combines a *Java* front-end with a mathematics-engine written in *Standard ML*, which reuses logical concepts and mechanisms of *Isabelle* [67]. Both projects are case studies for the use of functional programming in practical, interactive software projects which also demonstrate how the disadvantages of this paradigm's intrinsic properties like side effect free programming can be complemented by integrating functional and imperative programming. Analogous to *Isabelle*'s parallelization, *Isac* was reengineered for multi-core systems. The involved background knowledge, documentation and outcomes of this project form

the basis for this document. These topics will be expanded to include common parallelism and concurrency mechanisms suggested and implemented for other functional programming languages as well as previous research on refactoring techniques for multi-core.

1.2 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 establishes a theoretical background on the different areas relevant for the practical work on the case study within *Isac* and includes topics such as functional programming, computer theorem proving, parallelism, concurrency, responsiveness and refactoring of programs within the functional paradigm. Subsequently, various approaches towards parallelism that can and may have been used in functional programming are investigated in chapter 3. The end of this chapter briefly talks about refactoring of functional programs for parallelism. Chapter 4 is then dedicated to the two main software projects involved in the practical part of this thesis, *Isabelle* and *Isac*, as well as to a discussion of what has been done in the course of the practical work. A conclusion and outlook on possible future work are given in chapter 5. Appendix A provides code samples in addition to those given in the main text as well as comments and further explanations. Finally, appendix B lists the contents of the DVD that was handed in along with the thesis.

Chapter 2

Fundamentals

In this chapter we want to get an overview of fundamental concepts relevant for the practical project work carried out along with and as a basis for this thesis. We will also consider ideas and technologies that are related to the topic of the thesis and could be utilized for similar practical applications. Section 2.1 explores the history and common features of functional programming. Computer theorem proving will be introduced in section 2.2, followed by 2.3 on a theoretical background on parallelism. Sections 2.4 and 2.5 explain concurrency in relation to latency and responsiveness and refactoring of programs developed in functional programming languages, respectively.

2.1 Functional Programming

Functional programming is a declarative way of programming, i.e. source code does not describe routines but rather the desired result of a computation. Computation is the evaluation of expressions. Even programs are in fact functions; hence the name *functional programming*. Pure functional programming languages do not permit mutable data which means that there are no variables, only constant values. Functional programs are side effect free, i.e. given the same input parameters they must always produce the same results. Unlike subroutines in imperative languages, here the term *function* refers to a function in the mathematical sense. Therefore the implementation of programs tends to be closer to their specification compared to other paradigms. Although functional programming languages are rarely utilized for the development of commercial software [36], they have had a significant impact on the theory and pragmatics of other paradigms and programming languages in general [45].

2.1.1 History

The history of functional programming dates back to a time before computers existed, when Alonzo Church published his work on the lambda calculus in 1932 [20]. This formal system was the first methodical approach towards describing a computational perspective on mathematical functions. One of its most notable properties is function self-application, which gives the lambda calculus its power whilst maintaining its consistency as a mathematical system. One of the first languages which contained elements of what is now known as functional programming is *Lisp*. Its initial language design, established in 1958, was only slightly influenced by the lambda calculus, but later dialects have encouraged a more pure, side effect free programming style [61]. The original purpose for the design of *Lisp* was a language for **lis**t **p**rocessing to be utilized in artificial intelligence research. A notable feature of *Lisp* are higher-order functions which can be found in most contemporary functional languages and which enable list iterations without the need for explicit loops.

In 1978 John Backus, who had been heavily involved in the design of *Fortran* and *ALGOL*, held a Turing award lecture which emphasized certain benefits of functional style languages over imperative programming. The document which resulted from the lecture [8] strongly influenced and fueled research in the area of functional programming. Also in the mid 1970s, Robin Milner and others from the *University of Edinburgh* designed the *ML metalanguage* as a command language for the theorem prover *LCF* [62] (see section 2.2). They continued the development of *ML* and turned it into a stand-alone programming language. Even though it has impure features such as mutable references and an I/O system which is not side effect free, *ML* is known as a functional programming language because it encourages a functional programming style. *ML*'s type system employs the *Hindley–Milner type inference algorithm* (see section 4.1.1) which reduces the number of required type annotations because it can infer the types of expressions in most cases. Later, a standard for the language was established [63], which resulted in the programming language *Standard ML*.

As a dozen non-strict languages (i.e. employing lazy evaluation) emerged in the 1980s, it was decided at a conference in 1987 that the parallel efforts should be merged into one common language as a basis for an exchange of ideas. This decision resulted in the formation of a committee which created *Haskell* [75]. Research around this purely functional, general-purpose programming language has produced many innovations in the area of functional programming during the last two decades.

2.1.2 Common Features

There are certain distinguishing features which can be found in most contemporary functional programming languages. This section outlines the most prevalent ones (e.g., see [105, 45]).

Referential Transparency

The heavy use of immutable data has already been mentioned in section 2.1. A consequence of pure functions is *referential transparency*, which is the property of computations which always yield equal results whenever they are invoked with the same input parameters. Later in this section we will see how pure languages enable e.g. input, output and exception handling whilst preserving purity.

Recursion

Recursion is not a technique which is specific to functional programming. However, since loops require the use of mutable values (counter variables), recursion is used way more excessively than in imperative languages. Often, it is the only way to describe iterations. As recursion can easily cause an overflow of the call stack, most implementations of functional languages employ *tail call optimization*. This mechanism discards a function call's environment and thereby avoids creating a new stack frame for the recursive call and saves memory. This only works for true tail recursion, i.e. if the result of the recursive call is the result of the function itself.

Higher-Order Functions

Higher-order functions (HOFs) are functions which expect other functions as their input parameters or have functions as their return values. Examples for frequently used HOFs are the *map* and *fold* operations, which apply other functions to lists of values as in the *Standard ML* expression

```
val ls = map (fn x => x + 2) [1, 2, 5]
```

which would result in the list `[3, 4, 7]`. HOFs generally require languages to treat functions as first-class citizens, i.e. simple values. Although HOFs are a central aspect of functional programming, imperative languages may facilitate the use of functions as arguments of other functions. E.g. *C* allows the use of function pointers as first-class citizens. As we will see later, e.g. in section 3.2.3, HOFs can be beneficial for the expression of concurrency.

Purity and Effects

Many functional programming languages such as *Standard ML* encourage a purely functional coding style. However, certain functionality like memory

I/O require developers to write impure code. Purely functional programming languages like *Haskell* and *Clean* do not permit any side effects and therefore use other mechanisms to accomplish the desired effects. The *Haskell* developers decided to implement a language construct called *monad* [75]. Monads can be described as representations of possibly impure computations. In order to preserve purity, functions which perform e.g. I/O operations have to accept monads as arguments or return them. Similarly, *Clean* uses *uniqueness types* [1] which ensure that every function's type makes transparent to the environment, which mutable aspects of the environment it manipulates. This is accomplished by allowing these functions exactly one access to the denoted mutable data which they receive as arguments and whose manipulated versions are contained in their return values.

Partial Function Evaluation

Partial function evaluation facilitates applying less arguments to a function than it requires which results in another function that only requires the remaining arguments. E.g. the function `fun add x y = x + y` accepts two arguments. Its partial evaluation `val add2 = add 2` results in a function which only expects one argument and adds 2 to it.

Pattern Matching

Pattern matching allows for data to be processed on the basis of its structure. E.g. in *Standard ML*, functions are usually expressed with exhaustive pattern definitions. The classic definition of the Fibonacci numbers[1] would be

```
fun fib 0 = 0
  | fib 1 = 1
  | fib x = fib (x - 1) + fib (x - 2); .
```

Lazy Evaluation

Due to referential transparency, the order in which expressions are evaluated does not affect the results. Therefore, some languages like *Haskell* defer the computation of expressions until they are actually needed. This avoids unnecessary calculations and makes infinite data structures possible. On the other hand, lazy evaluation can potentially cause space leaks [53] and it makes formal reasoning about code hard [56]. The opposite is called *eager* or *strict* evaluation which is the standard for most imperative languages and e.g. *Standard ML*.

[1] see comments in appendix A.1

2.2 Computer Theorem Proving

Computer Theorem Proving is related to functional programming in several fundamental ways. My practical work for this thesis concerned TP (**T**heorem **P**roving) on a technical level below the logical level of TP, on the level of parallel execution on multi-core hardware. However, it is the logical level of TP which provides the fundamental relations to functional programming.

2.2.1 History

In order to understand TP, we start with the history of TP from the very beginning.

AI and *Lisp*. Mechanical or "formal" reasoning was an original interest of Artificial Intelligence (AI). The first important programming languages in AI were *Prolog* (logic programming) and *Lisp*. The latter was briefly introduced as a functional language in section 2.1.1.

Automath. A first notable success was the mechanization of Edmund Landau's *Foundations of Analysis* [10] in *Automath* in 1969. *Automath* is outdated, but some of its achievements are still used in TP, e.g. de-Brujin indices, the lambda calculus and Curry-Howard correspondence. This early success in TP was not acknowledged by mainstream mathematics. Rather, TP was mixed up with predictions of early AI, e.g. Herbert A. Simon's claim that "*[m]achines will be capable, within twenty years, of doing any work a man can do.*" [83]; predictions that soon turned out to be unrealistic. Nevertheless, mechanization of various logics (sequent calculus, temporal logics, communicating sequential processes (see section 3.4.3), etc.) was continued and led to many different, unrelated software tools.

LCF. *LCF* (Logic for Computable Functions) [62] denotes an interactive, automated theorem prover developed at the universities of Edinburgh and Stanford in the early 1970s. The *LCF* project is relevant for this thesis, because it introduced the *ML* programming language explored in section 4.1.1 and was used for the practical work within the thesis. The purpose of *LCF* was to allow users to write theorem proving tactics and to encapsulate theorems as abstract datatypes, such that only the admitted inference rules can establish a theorem. This is still called *LCF principle*. *LCF* became the ancestor of the *HOL family* of provers introduced below.

The Next Seven Hundred Theorem Provers. As mentioned above, although an academic niche, TP developed many different and unrelated software tools in the 1970s and 1980s. So in the 1980s the situation with TP was similar to the situation with programming languages in the 1960s.

In 1965 Peter Landin had published a talk and a paper titled *The next 700 programming languages* [52]. This paper influenced the convergence of programming languages and led to focused efforts on a few leading products. When Larry Paulson started the development of a generic logical framework with the purpose of allowing various logics to be implemented within this framework, he reused the analogy and published a paper titled *The Next Seven Hundred Theorem Provers* [73]. This paper addressed the general issues involved in the design of a TP, the many technicalities to overcome until a system becomes usable and warns to produce provers for one-time usage: "*Programs like* Isabelle *are not products: when they have served their purpose, they are discarded.*" [73]. In fact, the resulting logical framework, *Isabelle*, was able to attract efforts to implement new logics [102] within *Isabelle* and thus focused efforts, avoiding development of further TPs from scratch.

The HOL family of provers. This family of provers is a descendant of *LCF* introduced above and models higher-order logic, a specific logic which does not have sets (as preferred by mathematics) at the axiomatic basis, but functions. Thus this family shows great affinity to functional programming. The ancestor of this family is *HOL* [30]. *Isabelle* [69] abstracts from higher-order logics to natural deduction and thus allows for implementations of various logics within one and the same system as mentioned above. *HOL Light* [107] uses a much simpler logical core and has little legacy code, giving the system a simple and uncluttered feel for programmers. *Isabelle* is also a good example demonstrating that the traditional distinction between automated theorem proving (ATP) and interactive theorem proving (ITP) is obsolete today. *Isabelle* and other interactive proof assistants include various automated provers in order to assist interactive construction of proofs. *Isabelle*'s collection of automated theorem provers is called *Sledgehammer* (see section 4.1.5).

Breakthrough in mathematics: the four color theorem. The four color theorem was stated more than a hundred years ago, but it turned out to be hard to prove. In 1976 the theorem was tackled by Appel and Haken by the use of computer software [5]. However, they could not prove their program correct and thus their work was not acknowledged as a mathematical proof. In 2005, Georges Gonthier [29] used TP for proving the theorem and since that time TP was accepted as an indispensable tool for large and complicated proofs. Gonthier used the TP *Coq* [19], which works within the theory of the calculus of inductive constructions, a derivative of the calculus of constructions. Another famous theorem is the Kepler conjecture which took even longer to be proved. In 1998, Thomas Hales presented a proof of 250 pages of notes and 3GB of computer programs, data and results. In

2003, after four years of work, a prominent team of referees reported that they were "99% certain" of the correctness of the proof [33]. Since a mathematical proof is either correct or not, Hales started the *Flyspeck* project [104] in order to employ TP for a complete and trustable proof. The mentioned webpage hints at the important role of *Isabelle* in the project.

Continuous penetration of computer science. In computer science there are no spectacular breakthroughs like in mathematics. But there was much work on formalizing foundations from the very beginning. In the early days of computer science, Austria had a world-renowned group gathered by Heinz Zemanek [124]. At *Vienna University of Technology*, Zemanek had built one of the first computers only with transistors, the *Mailüfterl* in 1955. And IBM, the dominating company at that time, hired Zemanek to implement IBM's programming language number one, *PL1*. *PL1* was a monstrosity born by an IBM management decision to join *Fortran* with *COBOL* using insights from *ALGOL*. The overwhelming difficulties to implement *PL1* were solved by what later became known as the *Vienna Development Method* (VDM) [26], an ancestor of what is now called *formal methods*.

Formal methods, as the name indicates, formalizes a domain with mathematical notions. In parallel, computer science clarified the logical foundations of programming (Hoare, Dijkstra and others), such that programs could be proved correct with respect to a formal specification, i.e. proofs rely on specifications created by formal methods. Ever since the *Pentium bugs* [109, 81] swallowed several millions of dollars, Intel has invested in formal methods. For instance, Intel supports the TP *HOL Light* mentioned above. As there are many safety critical applications in medicine, air traffic control, etc. and technical systems become more complex, the demand for formal methods is increasing. This includes TP.

2.2.2 Relevance

The executable fragment of HOL and code generation. As already mentioned, higher-order logic is closely related to functional programming. This relation includes an executable fragment, i.e. equational theorems within HOL, which can model programs and which can be evaluated within the logic. This way of embedding programs in the logical environment provides the best prerequisites for proving properties of programs and for software verification. *Isabelle* recently implemented a *function package*. Execution, or more appropriately, evaluation, within a prover is very inefficient in comparison with state-of-the-art production software. Therefore engineering software cannot be built within a prover. However, it turned out that algorithms described within the executable fragment of HOL are a concise source for automatic generation of efficient code in a surprisingly straight forward manner.

Functional programming. Software verification tools are presently already more advanced for functional programming languages. And the theoretical background presented in this section indicates that verification tools will develop better in the paradigm of functional programming than in other paradigms. Also, efficiency considerations are expected to change in favor of the functional paradigm. Programs written on an abstract level in a TP can be transformed into efficient code by automated code generation. This transformation can potentially handle parallelization much more conveniently than programs developed within other paradigms [32].

2.3 Parallelism

Concurrency in software has been around for a long time and without it, multitasking would be impossible. It means that processes and threads may be started, executed and completed in overlapping time periods. This does not necessarily imply any parallel, i.e. simultaneous, execution. Parallel software is structured in a way which allows for literally simultaneous execution, e.g. on multi-core processors. Now that this kind of processors is used heavily in PCs, laptops and smart phones, efficient software must be able to exploit parallelism. Approaches to parallelism can roughly be grouped into three different levels: *instruction-level* (section 2.3.2), *task parallelism* (section 3.2) and *data* (section 3.3). The following sections will establish a theoretical background of parallelism.

2.3.1 Gain, Cost and Limits

Formulated in 1967, Amdahl's Law [4] describes the correlation between the inherently sequential fraction of a program and the resulting maximally possible speedup S_A under parallel execution (see fig. 2.1), given by

$$S_A(n) = \frac{1}{1 - f + \frac{f}{n}}, \tag{2.1}$$

where f is the ratio of the parts of a program which can be parallelized and n is the number of processing units. The upper limit for the speedup for a theoretically infinite number of processors is therefore

$$\hat{S}_A = \lim_{n \to \infty} \frac{1}{1 - f + \frac{f}{n}} = \frac{1}{1 - f}.$$

Amdahl's Law expects the input dataset size to be fixed. However, John L. Gustafson discovered that generally the fraction which can be computed in parallel behaves proportionally to the problem size, i.e. the bigger the input dataset, the higher the percentage of calculations which can be carried out simultaneously [31]. Therefore Gustafson's Law approaches the problem

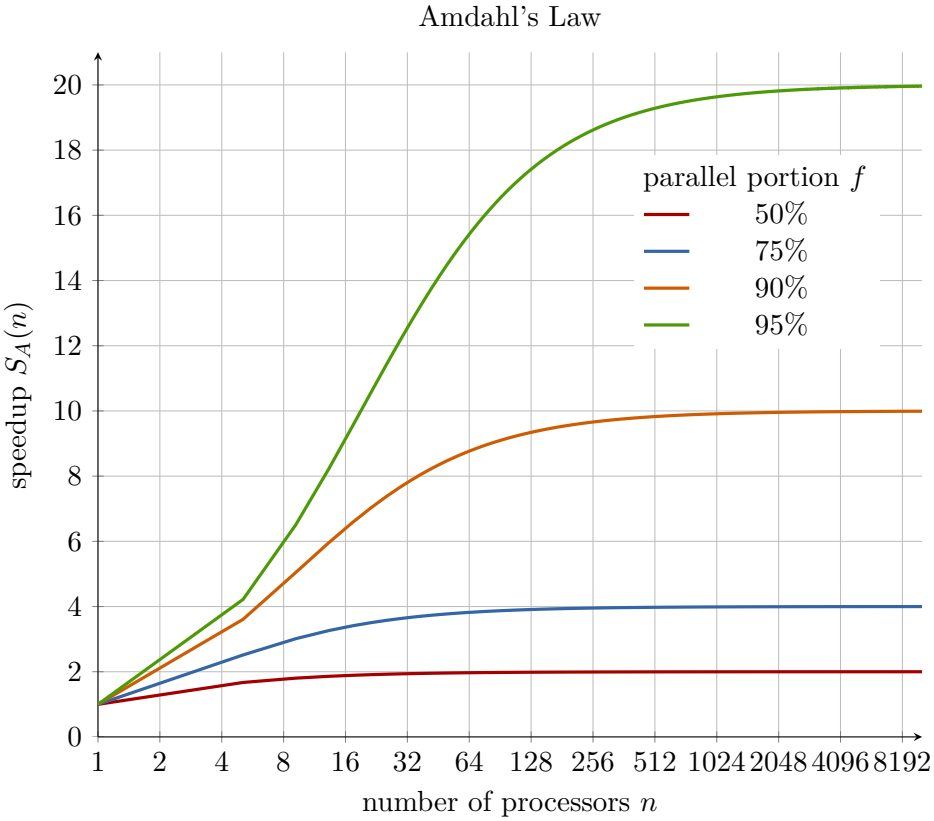

Figure 2.1: Amdahl's Law and potential speedups.

from a different perspective. It assumes that in practice an algorithm's problem size depends on available temporal and computational resources. The resulting *scaled speedup* S_G identifies the ratio between total and parallel execution time of an algorithm which linearly increases with the number of processors (see fig. 2.2) and it is given by

$$S_G(n) = 1 - f \cdot (1 - n). \tag{2.2}$$

The two theoretical speedups do not take into account dependencies or communications of parallel tasks. These factors can cause a performance overhead which may exceed the speedup if parallelization is taken too far. A certain class of parallel problems, called *embarrassingly parallel problems* [64], do not require any communication.

2.3.2 Multi-Core Processors

While the computing industry continued to fulfill Moore's predictions [65] by fitting more and more transistors on smaller chips and thereby produc-

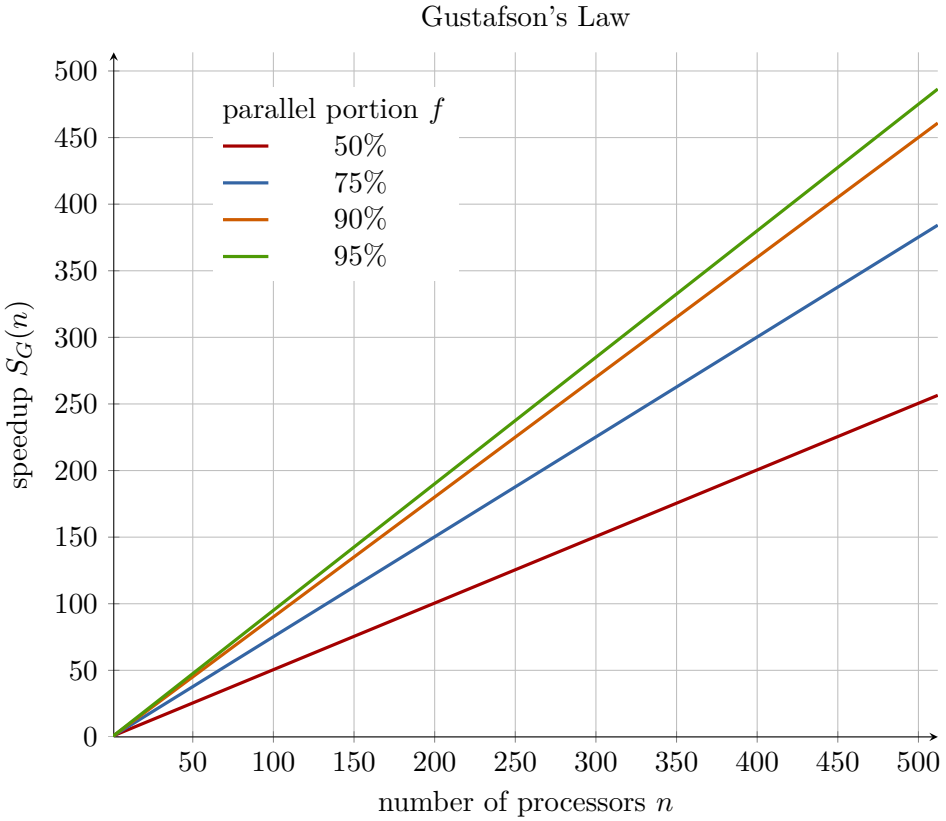

Figure 2.2: Gustafson's Law and potential speedups.

ing persistently faster uniprocessors, it is now physically almost impossible to make transistors any smaller [28]. Heat dissipation resulting from transistor density on the fastest chips requires extensive cooling and causes a high power consumption. Therefore hardware manufacturers have turned to building chips with multiple processing cores which may not be as fast as recent uniprocessors, but produce less heat and are more energy efficient. Architectures differ in the number of cores and their memory models. Several or all cores may share one cache or may each have their own cache. Separate caches avoid the challenges of coordinating cache accesses, but sometimes a centralized cache can be faster. Unlike multiple chips, several cores on the same chip can share memory management and memory elements and signaling between them can be faster because they are placed on the same die.

Multi-core processors, however, could not solve all performance issues, even if software were able to fully exploit their power. Other factors like memory bandwidth can turn out to be performance bottlenecks [66]. Also, these processors are not the only way hardware may support parallelism:

Bit-level parallelism takes advantage of increased processor word sizes in order to reduce the number of instructions that need to be executed to complete operations on variables which are longer than one word [21].

Instruction-level parallelism tries to discover data independent instructions which often can be reordered and computed during the same processing cycle. Also, instruction pipelines with multiple stages allow different instructions to be in different stages and thereby overlap their processing [41].

Simultaneous multithreading enables multiple threads to issue instructions on a single processing core which may be processed during the same cycle [90].

2.3.3 Operating Systems

Operating systems are heavily involved in the resource management of their underlying architecture, both for their own operations and for user applications. Besides support for simultaneous multithreading (see previous section), general-purpose OSes have to deal with diverse hardware approaches, e.g. multiprocessors vs. multi-core processors, shared vs. separate cache models. Scheduling algorithms and memory management are facing new challenges because they must balance loads between cores while avoiding cache misses and reducing the number of context switches. In this thesis we will assume symmetric multiprocessing (SMP), i.e. systems whose processing cores are identical and access the same main memory. Scheduling algorithms in OSes for more heterogeneous systems also need to take into account differing performance characteristics and possibly even instruction sets of several processors.

2.3.4 Functional Programming

Imperative programming lets the programmer make the steps that the system should carry out explicit. The idea of declarative, including functional, paradigms is a description of the desired result of a computation. On the spectrum between an algorithm's formal specification and the resulting machine code, functional code is generally said to be particularly close to the formal specification [40]. Therefore this high-level programming paradigm puts higher demands for optimization on compilers and run-time systems. This also includes the management and exploitation of multiple CPU cores.

As purely functional programming prohibits side effects, parallelizing such programs should be trivial. However, in practice, general-purpose functional programming languages are not only used for *embarrassingly parallel problems*. Harris and Singh [38] identify five issues which may arise when trying to automatically parallelize functional programs:

1. Inherent parallelism may be limited, i.e. call hierarchies may enforce a certain order of execution.
2. Parallel executions need to be managed by the run-time system. This introduces an overhead which can even cause a slowdown if the parallelizable work is too fine-grained.
3. Sometimes, purely functional code may contain side effects when it is compiled (e.g. memoization).
4. Purely functional general-purpose languages generally do have means of encapsulating side effects such as I/O or updates to mutable data.
5. In languages supporting lazy evaluation it is not known whether expressions actually need to be computed until their results are queried.

For more details see chapter 3 which is entirely dedicated to parallelism in functional programming languages.

2.4 Concurrency, Latency and Responsiveness

Concurrency can improve system performance by hiding latencies. While one thread performs blocking operations such as e.g. disk I/O, another thread's execution can continue to perform other calculations independently. In cases where latencies are introduced only because a computationally complex operation claims the processor for a long period of time, multithreading cannot improve performance. But it can improve responsiveness by updating the user interface, giving feedback about pending operations' status and react to user input. Responsiveness does not necessarily require high performance, but it is a major factor of the user experience that an application is able to deliver. Concurrency can facilitate increased responsiveness. However, reasoning intuitively as well es formally about concurrent software and developing it is said to be difficult, mostly due to undetermined execution orders and thread synchronization issues [85].

2.5 Refactoring Functional Programs

Refactoring is a process which changes the design and structure of existing software without changing its external behavior. It can result in simpler programs, remove duplicate code or prepare an extension of the program's functionality. Because refactoring is a potentially tedious and error-prone process [88], there is a variety of tools which automatically check preconditions for and apply refactoring steps for object-oriented programming languages. The number of such tools for functional programming languages, however, is limited. One of the few existing projects targets *Haskell* code and is called *HaRe* (***Ha****skell* **Re**factorer). It follows five main strategies:

1. **Generalization.** Extract some of a function's behavior into an argument, possibly a HOF, and thereby make the function more general and reusable.
2. **Commonality.** Similar or identical parts of a program are identified, extracted into a single function which is invoked at the respective locations and with appropriate parameters.
3. **Data Abstraction.** Replacing algebraic datatypes by abstract datatypes often allows programmers to modify concrete implementations without touching client code.
4. **Overloading.** The use of *type classes* and *instances* allows for overloading of names. This may improve readability and facilitate code reuse.
5. **Monadification.** Packing code into monads also makes it possible to separate parts of a system from the client code, i.e. developers can modify the monads without changing client code.

HaRe furthermore implements structural refactorings such as deletion of unused definitions and arguments, renaming of several kinds of program items and modifications of the scopes of definitions. Later versions introduced module-aware refactorings including import/export function organization and moving definitions between modules. For more details and refactoring techniques implemented in *HaRe* please refer to [88].

Type classes and *monadification* are specific to *Haskell*. In *Standard ML* overloading can be achieved simply by omitting the type indicators, thanks to the Hindley–Milner type inference algorithm. Please note that *Standard ML* is a strongly typed language [63].

Chapter 3

Parallelism in Functional Programming

This section presents several concepts for parallelism and concurrency which have been suggested for and implemented in functional programming languages. The central idea here is the philosophy of the declarative paradigm: The developer should not have to worry about *how* the parallel execution needs to be organized but at most make informed decisions on *what* has to be parallelized. Ideally, process management details such as process creation, resource assignment and synchronization should be taken care of by compiler and run-time system. In practice, various properties of languages such as impureness and monadic I/O, may allow for language constructs from imperative programming or even require forms of explicit synchronization.

Many of the mechanisms presented in the following sections have been implemented in and modified for certain programming languages. It seemed appropriate to place details on these realization details and changes in the respective sections. Table 3.1 gives a brief overview. Note that it only contains those mechanisms and sections that mention concrete implementations. For consistency reasons, all sample code is provided in *Standard ML*, even where the demonstrated concepts are not actually available in the language.

3.1 Implicit Parallelism

Due to the lack of side effects, it should be possible to compute independent subexpressions in parallel. This means that in theory, purely functional programs are parallelizable automatically by the compiler. However, the whole process is not trivial as section 2.3.4 already pointed out. A compiler that supports implicit parallelization must be able to perform granularity and cost analyses in order to achieve advantageous parallelizations and avoid those whose overhead would exceed their potential gain because they are too fine-grained. Additionally, compilers for lazy languages have to incor-

Table 3.1: Overview of parallelism mechanisms and referenced programming languages.

Concept	*Language*	*Section*
Annotations	*Clean*	3.2.2
Futures and Promises	*Multilisp*, *Isabelle/ML*, *Scala*	3.2.3, 3.4.2, 3.4.3
Parallel Combinators	*Haskell*	3.2.4
Algorithmic Skeletons	*Java*	3.2.5
Data Parallelism	*Haskell*	3.3
Software Transactional Memory	*C*, *Haskell*, *Java*, *JavaScript*, *Python*, *Scala*	3.4.1
Actors	*Erlang*, *Scala*	3.4.2
Communicating Sequential Processes	*occam*, *Limbo*, *Go*	3.4.3

porate a strictness analysis, i.e. identify expressions which must necessarily be evaluated, such that they do not introduce an overhead by computing expressions whose results are not needed. An example for a function with a strict argument is the `map` operation. If its list argument is undefined, the result of the whole function call will be undefined.

3.2 Task Parallelism

Instead of automating parallelization, functional general-purpose languages such as *Haskell* and *Clean* provide special syntax to indicate that expressions are to be evaluated in parallel and thereby make theoretically implicit parallelism explicit.

3.2.1 Parallel `let`-Expressions

One proposed method are parallel `let`-expressions [35]. `let`-expressions are a common way in functional languages to allow the programmer to declare a list of values required for the evaluation of one final expression. The modified version `letpar` has been suggested for the indication of a list of independent values which could potentially be computed in parallel. While this keyword is not available in *Standard ML* a hypothetical use could look like the expression

```
letpar
  val x = add2 3
  val y = fib 8
  val z = add 7 5
in x + y + z end; .
```

3.2.2 Annotations

Another straight forward solution is the use of annotations. This is probably the simplest way of expressing parallelism because it does not affect the semantics of a program but only its runtime behavior. By ignoring the annotations the program can then still be compiled sequentially. This method has been implemented in the programming language *Clean* [70] which also allows indications of the target processing unit for the evaluation. Since eager languages evaluate a function's arguments before invoking the function itself, this mechanism, in its simple form, only works for lazy languages. A potential implementation of the parallel `fib` function[1] in a lazy version of *Standard ML* could look somewhat like the definition

```
fun fib 0 = 0
  | fib 1 = 1
  | fib x = fib (x - 1) + par (fib (x - 2)); .
```

A similar construct for eager languages is slightly more involved and will be discussed in the following section.

3.2.3 Futures and Promises

The concepts of *promises* and *futures* were first suggested in the mid 1970s [9]. The idea is similar to annotations (see previous section). One of the first languages that adopted futures was called *Multilisp* [47]. As the *Isabelle* theorem prover provides its own implementation of futures [60] which have been introduced to *Isac* during the parallelization process (section 4.3.2), they are of special interest for this thesis. Futures are objects representing the result of an expression which may not be known initially because its computation is managed by the run-time system and may occur in parallel. *Implicit* futures are usually an integral part of a language and treated like ordinary references. If results are not available the first time they are requested, program execution stops until the future value has been obtained. In contrast, *explicit* futures require programmers to manually enforce their computations before they can access their results. *Isabelle*'s solution follows the latter approach and will be discussed in more detail in section 4.1.1. The definition

```
fun fib 0 = 0
  | fib 1 = 1
  | fib x = let
        fun fib' () = fib (x - 2)
        val fibf = Future.fork fib'
      in fib (x - 1) + (Future.join fibf) end;
```

[1] see comments in appendix A.1

demonstrates the use of futures in *Isabelle/ML*[2]. The function `Future.fork` accepts one argument which must be a function of type `unit -> 'a`, i.e. one which accepts an empty argument "`()`" and returns an arbitrary type. `Future.join` requires a future value as its argument and returns the result contained in it.

As with general evaluation strategies, futures can be determined eagerly or lazily, i.e. their computation can be started on creation of the datastructure or on demand. They are read-only placeholders. In contrast, promises are the single assignment containers which hold the future's value. Futures and promises are closely linked to Communicating Sequential Processes (section 3.4.3) and also to the Actor model (section 3.4.2) and are therefore available e.g. in *Scala* [106]. Although their adoption originated in functional programming, their use is not limited to this paradigm.

3.2.4 Parallel Combinators

Haskell's approach to task parallelism is the use of parallel combinators. Its run-time system manages a queue of tasks, so called *sparks*, whose evaluation occurs as soon as an idle CPU is detected. The language provides the keyword `par`, which accepts two arguments, "sparks" the evaluation of the first and returns the computed result of the second. For more details see e.g. [57].

3.2.5 Algorithmic Skeletons

The idea of algorithmic skeletons is simple and not limited to parallelism. Most parallel algorithms show common patterns. These patterns can be abstracted into higher-order functions. A well known example is *divide-and-conquer*:

```
fun dac is_trivial solve divide conquer x =
    if is_trivial x then
      solve x
    else
      divide x
        |> map (dac is_trivial solve divide conquer)
        |> conquer; ,
```

where `x |> f` is syntactic sugar for `f x` and `is_trivial` is a function with one argument and a `bool` return value. In order to parallelize the algorithm we only need to use a parallel version of `map`. Other common patterns include *branch-and-bound* and *dynamic programming*. While this method is also available for imperative languages such as *Java*, the fact that functional languages treat higher-order functions as first-class citizens makes

[2] see comments in appendix A.1

them particularly suitable for the use of skeletons. Algorithmic skeletons hide the orchestration between sequential portions of code from the programmer. Sometimes the compiler can provide low-level support for the skeletons. Furthermore, multiple primitive skeletons can be composed to produce more powerful patterns. Unlike other high-level parallelism mechanisms, they are apt to cost modeling [36].

3.3 Data Parallelism

Many operations require the application of the same function on multiple items of a data set. This can generally occur in parallel. With this kind of algorithms, the parallel portion naturally depends on the problem size. Here the same issues apply that we encountered before: The granularity often is too fine and the overhead exceeds the benefits of parallel processing. Most applications of data parallelism are based on array structures and early implementations specialized in highly parallel, high-performance and scientific computing [36]. Recent developments in GPU hardware and the arrival of general-purpose GPU programming thanks to projects such as *CUDA* and *Accelerator* [86] have fueled data parallelism research during the last few years [68]. Distributed, parallel algorithms like Google's *MapReduce* should be mentioned here. However, we will not discuss distributed parallelism in this thesis.

In 1990, Blelloch and Sabot first made a distinction between *flat* and *nested* parallelism [12]. Until then, data parallelism implementations had only considered sequential functions for parallel application on data sets. Nested data parallelism is significantly more complex in that it allows the application of functions which themselves are parallel. More than 15 years later, this functionality was suggested and implemented as an extension to the *Haskell Glasgow Compiler* called *Data Parallel Haskell* [74].

3.4 Concurrent Functional Programming

As we saw in section 2.3, concurrency describes two or more tasks whose lifespans may overlap while parallelism is capable of exploiting multiple processing units by executing multiple tasks simultaneously. Section 2.4 outlined how concurrency can be beneficial by hiding latencies and improving responsiveness. Now we want to explore concurrency models and control mechanisms in functional programming.

3.4.1 Software Transactional Memory

While hardware support for transactions had been introduced before and they were a fundamental aspect of fault tolerance in the context of database

systems, software-only transactional memory was first suggested by Shavit and Touitou in 1997 [82]. Unlike *lock-based synchronization* STM generally is a non-blocking strategy. Transactions are sequences of read and write operations on a shared memory. They are tagged atomic blocks, processed optimistically, regardless of other threads potentially performing other alterations to the memory simultaneously. Each access to the memory is logged and once a transaction is finished, a final validation step verifies that no other thread modified any of the elements which were read during the process, i.e. the transaction's view of the memory was consistent. If there was no conflicting write access, the transaction is committed to the memory. Otherwise the whole process is repeated until no more conflicts arise. In order to be correct, transactions must be linearizable, i.e. it must be possible to execute them in a certain sequence without temporal overlaps, such that the resulting state of the shared memory is the same as if the transactions had been carried out concurrently [42]. Because of their non-blocking nature, STMs do not cause deadlocks or priority inversion. They are inherently fault tolerant to a certain extent, because transactions can be undone in the event of a timeout or an exception. Also, they are free from the tradeoffs between lock granularity and concurrency.

Recent work on *commitment ordering* has helped reduce the number of transaction conflicts and thereby improve performance of STM implementations [77, 101]. Also, there are various STM solutions that do in fact use locking mechanisms during different phases of transaction. They can potentially reduce the number of rollbacks and consequently also make process coordination faster [25]. The significant difference between these approaches and conventional, mutex based thread synchronization mechanisms is the fact that with STM, any locking is invisible to the programmer and taken care of by the run-time system. The downside of STM is the performance overhead caused by transaction management which typically impairs the performance of programs with a low number of threads, such that they are slower than when implemented in a sequential manner. STMs, however, scale particularly well with respect to higher thread and processor numbers [80].

Several implementations for various programming languages including *C*, *Java*, *Scala*, *Python* and even *JavaScript* are available. *Concurrent Haskell* offers multiple language constructs built on the basis of STM to allow for composable, nested, alternative and blocking transactions as well as STM exceptions [39].

3.4.2 The Actor Model

In 1973, Hewitt, Bishop and Steiger suggested a modular actor formalism [43] which laid out the basis of what is now known as the *Actor model*. Several publications of the following decade formed a comprehensive Actor model theory. The two central concepts of the Actor model are *actors* and

messages [2]. Actors are entities that can send and receive messages. Their behavior is defined by the **actions** they perform on receipt of a message (e.g. create another actor) and a finite number of other actors they know about, called **acquaintances**. The acquaintance relation is not necessarily bidirectional, i.e. an acquaintance B of actor A must not necessarily know actor A. Each actor has an immutable, unique name and a local state. Communication between actors is asynchronous. Incoming messages are processed one at a time and as a response to each message an atomic sequence of actions is performed. Since actors and messages are autonomous, encapsulated entities they are easy to reason about and can be composed into more complex systems [3]. The order of actions affects an actor's external behavior. But because messaging occurs asynchronously, the processing order of messages is undefined. This leads to non-determinism in computations based on the actor system. There are certain abstractions available which allow for constraints on message ordering and can therefore eliminate part of the non-determinism [48]. They can be summarized in four important properties:

1. **Encapsulation.** Unlike shared memory thread models, actors are highly encapsulated. Therefore, they provide no locking or synchronization mechanisms for shared resources. Their internal states are not directly accessible by other actors. The only way for them to exchange data or influence each other is by means of messages. This limitation, however, is violated by some implementations in languages such as *Scala* [89]. In programming languages which permit mutable data, messages should, in theory, not transfer references to locations in a shared address space. Otherwise the receiving party could modify memory locations that are contained in another actor's state representation. However, *Java*'s actor library *Akka* allows actors to do just that and it is the programmer's responsibility to maintain encapsulation.
2. **Fairness.** Another important property is fairness with respect to actor scheduling: a message will be delivered to its target actor eventually, i.e. unless an actor dies or displays faulty behavior, it will be scheduled in a fair manner.
3. **Location Transparency.** This property demands that an actor's unique name is independent of its location, i.e. its assigned CPU and memory can potentially belong to a remote machine and even be migrated at run time.
4. **Mobility.** The ability for this migration can enable load balancing and improved fault tolerance. However, implementations do not necessarily satisfy these properties (e.g. *Scala*). *Weak mobility* allows code or initial states to be moved, while *strong mobility* additionally supports execution state migration [27]. It has been shown that the presence of mobility can significantly improve an architecture's scalability [72].

To demonstrate the use of a theoretical actor library for *Standard ML*, let us define a trivial function `greet` and execute it in the background:

```
fun greet name = print ("Hello " ^ name ^ "!\n");
val greeter = Actor.create greet;
Actor.send greeter "John Doe";
```

Scala

Scala is a programming language for the *Java Virtual Machine* that combines concepts of object-oriented and functional programming [71]. Its distribution comes with a toolkit for actors and message passing functionality called *Akka*. *Scala*'s standard actors library up to version 2.9.2 [114] has now been deprecated and replaced by *Akka*. Its implementation is based on *Erlang* like, fault tolerant actors and its execution model follows an *event-based* approach, thus introducing *inversion of control*. Compared to *Scala*'s original actors, *Akka* actors are more performant, fault tolerant and offer automatic load balancing [87]. They draw on the language's functional programming features such as partial functions and, like *Erlang*, pattern matching which are convenient for message passing. Actors potentially provide safer concurrency than traditional shared memory mechanisms because by design, no race conditions can occur. Also, actors are more lightweight than threads.

Futures

As mentioned in section 3.2.3, futures are related to the Actor model. *Akka* provides a `Future` class whose constructor accepts a block (`{...}`) *b* and returns immediately with a future value which is a placeholder for the computation's result. *b* is then executed asynchronously by a thread pool managed by the run-time environment [106]. Instead of creating futures directly, *Akka*'s actors have a method `?` that one can use to send a message. The method then returns a future value whose calculation can be synchronized and retrieved using `Await.result`. There are further functions for composition, combination or ordering of futures as well as exception and error handling. The library also supports direct access to `Promises`, the containers that complete futures.

3.4.3 Communicating Sequential Processes

CSP (Communicating Sequential Processes) belong to the family of *process calculi* and are another model of concurrency. Process calculi are of special interest for functional programming in that they support a declarative notion of concurrency. CSP were first proposed in an acclaimed paper by Hoare in 1978 [44] and originally designed for reasoning about concurrent

processes. Like the Actor model they utilize message passing for communication. Implementations include *occam* [46], *Limbo* [24] and *Go* by Google [76]. The main difference from the Actor model is the use of a concept named *channels* in CSP. Actors communicate directly with each other while CSP send and receive messages on directed channels. Another distinction is the fact that in the original CSP model, messages are delivered synchronously and instantaneously. This means that when a process sends a message to a channel, it waits until another process has received it and the receiving party blocks until there is a message available on the channel. Processes are anonymous and need to establish channels between each other in order to be able to communicate which involves a rendezvous, i.e. they need to exchange receiving and sending ends of channels to allow them to communicate. While the basic proposal suggested point-to-point channels, this can also be extended to allow multiple processes to know and use a channel's sending and/or receiving end. The developers of *Go* have chosen an approach that differs slightly from the original CSP model: Channels are asynchronous and buffered. They are declared to transmit a specific datatype. Interestingly, channels are first class values. As a consequence, they can also be transmitted via channels. Using *Poly/ML*'s thread library, we can utilize a supposed CSP library to achieve the same behavior we already saw with actors and invoke `greet` asynchronously:

```
fun greet name = print ("Hello " ^ name ^ "!\n");
val (c_in, c_out) = Channel.create () : string channel;
fun greet_csp () = Channel.read c_out |> greet;
Thread.fork (greet_csp, []);
Channel.write c_in "John Doe";
```

Futures

CSP, too, can simulate futures (section 3.2.3): A future is a channel that can transmit exactly one element and the according promise is another process that sends the result of a computation to the channel.

3.5 Refactoring for Parallelism

Only few publications have been dedicated to refactorings for parallelism [23, 49, 100], even less so in the context of functional programming. The *ParaForming* tool for *Haskell* [13] appears to be the first attempt, followed by very recent work on *Erlang* [15] and language independent approaches [14, 37]. Due to the trend towards multi-, many- and even megacore machines, modern programming languages should support parallelism by design because the introduction of parallelism should not occur as an afterthought during later stages of the software development cycle. *ParaForming* enforces

a separation of concerns between business logic and implementation details usually taken care of by system programmers. It offers several additional refactorings for *HaRe* (section 2.5) to support efficient, coarse-grained task and data parallelism.

Introduce Task Parallelism. This refactoring takes an expression x from a function, embeds x into an `Eval` monad, sparks its evaluation using parallel combinators (section 3.2.4) and substitutes the occurrences of x for the sparked computation x'.

Introduce Data Parallelism takes an expression l which must be a list, replaces it by a `parList`, i.e. a parallel implementation of a list supported by the standard library, and ensures the parallel versions of operations on l.

Introduce Thresholding allows for the definition of limits to the granularity of the aforementioned refactorings. Thresholding requires a designated expression t and a threshold. Calls to the respective function are only evaluated in parallel if t is not lower than the threshold.

Clustering refers to data parallelism and accepts a minimum chunk size for a parallel list. Whenever the size of a list falls below this value, the sequential versions of operations are used.

For other refactorings, examples and a discussion of performance gains and overheads please refer to [13]. *ParaForming* is very specific to *Haskell* and its refactorings are relatively straight forward. However, they avoid common pitfalls like forgetting to substitute the use of an expression. Decisions on granularity are not made by the system, but their implementation is automated as much as possible.

Wrangler

In 2013, Brown et al. documented their work on an extension for a refactoring tool for Erlang, named *Wrangler* [15, 55]. The provided refactorings make use of skeletons (section 3.2.5) and thereby facilitate informed decisions on which refactorings and skeletons to use based on cost modeling.

Chapter 4

Isabelle and *Isac*

The case study described in this chapter was performed as an invasive development on *Isac*, a prototype of an educational tool for applied mathematics which is built upon the theorem prover *Isabelle* in order to ensure the logical correctness of calculations. The development was invasive in that it addressed basic mechanisms of *Isabelle* which are reused by *Isac*. The motivation for this development was to improve *Isac*'s integration with *Isabelle* as well as *Isac*'s efficiency and responsiveness in a multi-user setting. Both improvements are related to the introduction of parallelism in *Isabelle* during the last years.

Section 4.1 introduces *Isabelle*, its basic architecture and the main components it is comprised of. It furthermore discusses *Standard ML* and the *Poly/ML* compiler it uses (section 4.1.1). Section 4.1.6 is about *Isabelle*'s development and documentation workflow which has mostly been adopted for the work on *Isac*. An overview of the basics of why parallelism was added to *Isabelle*'s computation model in section 4.1.7 wraps up the main part about *Isabelle*. In order to make the gain from *Isac*'s improvements comprehensible, section 4.2.1 gives some background information about *Isac*'s principal design and architecture. Section 4.2.2 shows *Isac*'s architecture, which had to be adhered to by the study's implementation work. Since the case study concerns deeply invasive development into *Isac*, section 4.2.3 is dedicated to the question, how *Isac*'s prototype development relates to *Isabelle*'s ongoing development, which is invasive, too. The detailed descriptions of the study's implementation work are section 4.3.1 on the integration of *Isac*'s knowledge definitions into *Isabelle*'s parallel theory evaluation and section 4.3.2 on the introduction of concurrency to *Isac*'s user session management.

4.1 *Isabelle*

Isabelle is one of the leading computer theorem provers as introduced in section 2.2. It is used in many academic courses, courses on semantics of

programming languages, on program verification, on introduction to mathematical proof, etc. But the number of professional users can hardly be estimated. Continuous reading of the mailing list for technical support [117] suggests more than a hundred persons worldwide who already did a notable development within a PhD, some academic postdoctoral work or as engineers in industry. *Isabelle* represents more than hundred man years of development, dedicated within the last twenty five years and still ongoing. *Isabelle* was started by Larry Paulson as a "logical framework", an answer to the issue of *The Next Seven Hundred Theorem Provers* mentioned on page 8.

Today, development is ongoing at three universities: At the *Computer Laboratory of the University of Cambridge*, Larry Paulson develops specific automated provers, formalizes mathematics and security protocols. *The Chair for Logic and Verification* at the *Technical University of Munich* spawned from the institute of Manfred Broy and is led by Tobias Nipkow. Nipkow is responsible for the core system as well as for general theory development. About ten postdocs and PhDs are employed for these purposes [118]. The parallelization efforts and development of the *Isabelle/jEdit* frontend (section 4.1.4) are undertaken at *Université Paris-Sud* in the *Laboratoire de Recherche en Informatique*, supervised by Makarius Wenzel. Confirmed by a wide range of expectations in TP technology as mentioned in section 2.2, *Isabelle* strives for usability at the workplace of computer scientists, of practitioners in various engineering disciplines and of mathematicians. E.g. at *RISC Linz* there is an initiative for verified computer algebra using *Isabelle*. Usability issues were the driving force to introduce parallelism to *Isabelle*.

Isabelle integrates various tools and languages. This section describes the technologies involved in the system, its most important components and how they work together. Also, it discusses the repository, development and documentation workflow and the introduction of parallelism. The distribution for all major platforms can be downloaded from the *Isabelle* website [108]. Whenever files are referenced, "~~" stands for the distibution's root directory. These references address the *Isabelle* release *Isabelle2013-2*.

4.1.1 *Isabelle/ML*

This section presents *Isabelle/ML*, which allows programmers to embed *Standard ML* code into the *Isabelle* environment. The used compiler is called *Poly/ML*. It is fast and efficient, implements the full *ML* standard [63] and provides additional features like a foreign language interface and support for POSIX threads based concurrency. *Isabelle/ML* furthermore includes a number of libraries built on top of *Poly/ML* with extra language constructs for concurrency. Before we take a deeper look into these we need to discuss *Standard ML* and the *Poly/ML* compiler.

Standard ML

ML stands for *meta language*. The language was briefly introduced in section 2.1.1. It is a strongly typed, functional programming language, which is not pure as it allows side effects and updatable references. Many people contributed ideas to the language. In order maintain a common ground, a standard was established [63]. All the code samples in this document are written in *Standard ML* syntax. Since *Standard ML* is only a language definition which is not dictated by a reference implementation, there are multiple compilers available, most notably *Standard ML of New Jersey* (*SML/NJ*) [116], *MLton* [91] and *Moscow ML* [115]. They differ in their priorities such as optimization, standard library size or platform support. *ML* was originally designed as a command language for the interactive theorem prover *LCF* (section 2.2.1) [62]. Because of its history in this area, it was specifically suitable for the development of *Isabelle*. Inference rules can easily be implemented as functions that accept a theorem and return a new one. Complete theorems can be constructed by applying a sequence of rules to other known theorems.

The Hindley-Milner Type System. *Standard ML* makes use of *Hindley-Milner type inference* [62] based on a type system for the lambda calculus which utilizes parametric polymorphism, i.e. types can be declared using type variables (e.g. *Synchronized* signature on page 30) and the decision on the instantiated types is left to the inference algorithm. This allows for the generic definition of functions while still ensuring static type safety. The type inference method is complete and is able to always deduce the most general type of an expression without requiring type annotations. If it cannot find a suitable type for an expression, its declaration must contain errors. Because type checking in *ML* is secure, deduction of theorems is always sound. Wrong applications of rules lead to exceptions.

Another important feature of *Standard ML* is its design for robust, large scale, modular software. Signatures are part of the type system and describe the interfaces between different modules.

Poly/ML

In the course of his PhD thesis, David Matthews from the *University of Cambridge* developed and implemented a language called *Poly*, which had many ideas in common with *Standard ML*. The compiler was reimplemented in *Poly* itself. Later, Matthews rewrote parts of the compiler to compile *Standard ML* code. The result was *Poly/ML* [59] which is now available under an open source license. It implements the full *Standard ML* specification and provides a few extensions. The run-time system was written in *C*. *Poly/ML*'s most notable feature are its concurrency mechanisms. While the *Standard*

ML specification [63] is limited to sequential programming, *Poly/ML* provides concurrency primitives based on POSIX threads. Its native support for system threads and parallel garbage collection make it unique amongst *Standard ML* compilers. Another related project is *Concurrent ML*, an extension of *SML/NJ* [78]. The primitives in *Poly/ML* include threads, mutexes and condition variables. An example for forking a thread was shown in the communicating sequential process example on page 25. For details on the operations and their implementation, please refer to [60].

Isabelle/ML

Isabelle/ML is a way of programming in *Standard ML* which is encouraged and enhanced within the *Isabelle* infrastructure. It adds many library modules to plain *Poly/ML*.

Antiquotations. A very powerful mechanism for easy access to *Isabelle* system values and logical entities are so called *antiquotations* [98]. Depending on their particular purpose they are resolved at compile, link or run time. They can refer to various datatypes such as *theorems* (derived propositions), *theories* (containers for global declarations), *contexts* (deductive environment states) and many other concepts within the *Isabelle* framework. E.g. `@{context}` would refer to the context at compile time at the location where it is used, i.e. a *Standard ML* value of type `context`.

Concurrency. *Poly/ML* comes with very general mechanisms for concurrency. Its approach is based on shared resources and synchronized access. While this concurrency model is well established, it is easy for the programmer to overlook some detail in the coordination of threads and deadlocks can occur. In order to ensure correctness by design, *Isabelle/ML* provides higher-order combinators and thereby hides synchronization details.

Guarded Access. Guarded access primitives operate on an abstract type called `var`. They are available from the module *Synchronized*[1] and the most notable elements of its interface are declared as

```
type 'a var
val var: string -> 'a -> 'a var
val value: 'a var -> 'a
val guarded_access: 'a var -> ('a -> ('b * 'a) option) -> 'b
val change: 'a var -> ('a -> 'a) -> unit .
```

This type is merely a wrapper for an ML reference. Please note the type parameters `'a` and `'b`. The constructor creates a `var` from such a reference.

[1] `~~/src/Pure/Concurrent/synchronized.ML`

`value` is the inverse operation and performs a read access. Machine code produced by the *Poly/ML* compiler generally treats memory cells as volatile, i.e. calls to `value` need not be synchronized because read operations always return *some* value. The `guarded_access` combinator is where the magic happens. Besides the `var` element x with value v that the write access will be performed on, it expects a function f that maps v to an option element of the touple type *'a * 'b*. The type *option* is *Standard ML*'s equivalent of the `Maybe` monad in *Haskell*: It can either be `NONE` or contain a value, i.e. in our case `SOME (`y`,` $v2$`)`. f is executed in a critical section of x on its value v. As long as it returns `NONE`, the current execution context waits for another thread to change v. If f returns `SOME (`y`,` $v2$`)`, the value of x is set to $v2$ and the call to `guarded_access` returns with result y. The original definition of `change`[2], i.e. an unconditional, synchronized update to a `var` element is given by

```
fun change_result var f = guarded_access var (SOME o f);
fun change var f = change_result var (fn x => ((), f x)); .
```

It is easily possible to implement higher-order mechanisms using these primitives. See [60] for an example of a message queue.

Futures. While the last approach allows for process synchronization on a high level, most of the complexity persists and as software gets larger it still gets increasingly hard to coordinate a high number of concurrent computations efficiently. For this reason, *Isabelle/ML* provides *futures* (section 3.2.3) and is thereby able to completely hide threads and their synchronization from the programmer. The goal was to provide a mechanism for parallelism which is both simple to use and performant with a special emphasis on operations typically carried out by a proof engine. *Future values* fulfill these requirements. They are value-oriented, i.e. the focus hereby is on parallelism rather than concurrency. Error handling happens synchronously by means of exceptions and asynchronously with interrupts. The *Future* module[3] has a comprehensive signature. The following declarations are only an excerpt:

```
type 'a future
val fork: (unit -> 'a) -> 'a future
val cancel: 'a future -> unit
val is_finished: 'a future -> bool
val join: 'a future -> 'a
val map: ('a -> 'b) -> 'a future -> 'b future
val value: 'a -> 'a future
```

[2] `~~/src/Pure/Concurrent/synchronized.ML`
[3] `~~/src/Pure/Concurrent/future.ML`

Notice that `'a` is a placeholder for an arbitrary type which a future can represent. Calls to `Future.join` synchronize the evaluation and wait for the future's result if it is not already available. In case an exception is raised during the evaluation, its propagation will wait until the future is joined. `Future.cancel` stops a future's evaluation if it has not finished, otherwise the function has no effect. The evaluation of futures is managed by means of tasks and worker threads. The number of workers should be related to the number of available processing cores. The exact number, however, is adjusted dynamically because the future scheduler always keeps a limited number of inactive threads in memory, such that they can immediately take over if another thread is temporarily stalled. In practice, the additional functions `Future.map` and `Future.value` have turned out to be useful in reducing the number of tasks and as a consequence the scheduling overhead. `Future.map` can append another operation to an existing future, which will be executed on the originally scheduled function's result. If the original future's evaluation has not started yet, the appended operation will be computed within the same task. `Future.value` produces a dummy future holding the function's argument as a result. Additionally, futures can be assigned priorities in order to influence execution order which by default is based on creation time. Also, futures can be organized in groups and even hierarchies of groups. This allows futures that depend on each other to be canceled whenever one of them raises an exception.

A simple *future* example has already been shown on page 19. Internally, the *Futures* module is based on the guarded access mechanisms introduced above. Based on futures, *Isabelle/ML* provides more specific parallel combinators such as the parallel list combinators `map` and `find`. The latter makes use of future groups and throws an exception as soon as a match is found, such that other search branches are not evaluated. This example shows how exceptions can be used as a basic means for functional interaction of futures without reintroducing the complexities usually associated with inter-process communication. See [60] for further details and a discussion of the performance of futures.

4.1.2 *Isabelle/Isar*

While in older versions of *Isabelle*, theorem proving tasks had to be solved using raw *ML* tactic scripts, theories and proofs can now be developed interactively in an interpreted, structured, formal proof language called *Isar* (Intelligible semi-automated reasoning) [96, 99]. *Isar* was designed to address an issue common to state-of-the-art theorem provers: Despite successfully formalizing a good amount of existing mathematical knowledge, logics and computer science theory, it was still hard to reach potential target groups unfamiliar with the involved computer programming requirements. The primary users of theorem provers should be mathematicians and engineers who

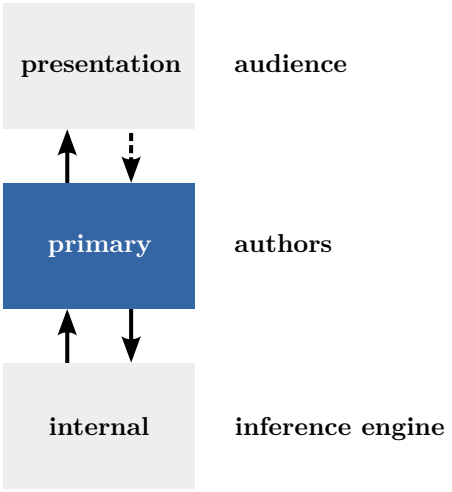

Figure 4.1: *Isar* proof notions [96].

utilize them to formulate new proofs and for verification purposes. *Isar* enables these users ("authors" in fig. 4.1) to write proofs and theories in a way that can be understood by machines while being much closer to human language than *ML* code or some logical calculus. As much as possible, it hides operational details thanks to its notion of *obvious inferences*. With a little extra care it can therefore also be used for presentation to an audience. The short, well-known proof that the square root of 2 is an irrational number is presented in appendix A.3.

The *Isar virtual machine* interpreter performs proof checking and incorporates an operational semantics for *Isar*. It is provided by the *Isabelle/Isar* framework, which embeds Isar into the *Isabelle* environment. It supplies the *Isar/VM* interpreter with all available proofs and definitions which are implemented in theory documents. These theories are structured in an inheritance hierarchy that forms a directed, acyclic graph. *Isar* proofs operate on the basis of certain object logics. The most established logic is *Isabelle/HOL*, i.e. higher-order logic (simply typed set theory), but a wide range of logics is available through the *Isabelle/Pure* meta logic. Another important element are *proof methods*, which are function mappings between proof goals and according proof rules. *Isar*'s lack of mechanisms for the formulation of automated proof procedures turned out to cause an unacceptable amount of duplicate code. Very recent work on a new language called *Eisbach*, which is based on *Isar*, has tried to address these issues [58].

4.1.3 *Isabelle/Scala*

The original user interface for the *Isabelle* theorem prover was a project called *Proof General*. It is a proof assistant front-end based on *Emacs*, also

available for other provers like *Coq* [19]. Its interaction model offers some improvements over a basic TTY model. The user enters a prover command and immediately receives a result, such as information on proof state or goals, before they can issue the next command. But interaction remains sequential and synchronous. There are two main weaknesses to *Proof General* and similar approaches. The underlying editor framework is outdated: *Emacs* offers a very powerful environment, but its graphical user interface can be considered dated. Also *Emacs*' *Lisp* engine does not support multithreading and does therefore not allow the development environment to make full use of the available CPU power. The other main shortcoming is the poor interaction model that is limited to a synchronous input / output sequence. Instead of simple text editing tools, a fully-featured IDE was desirable [92]. In addition, the IDE should be available on many platforms and as the main development focus of *Isabelle* developers is theorem provers, they wanted to be able to draw on existing IDE frameworks, rather than developing a new one from scratch. They chose to make use of the *Java Virtual Machine* and enable tools written for the *JVM* to communicate with the *Isabelle* prover via simple byte streams. The resulting asynchronous document model utilized for the communication between *ML* and *JVM* allows GUIs, text editors like *jEdit*, IDE frameworks such as *Eclipse* as well as web services to make use of the *Isabelle* prover. The outcome of the efforts to design this **p**rover **IDE** framework is called *Isabelle/PIDE*. In order to overcome the differing programming models and datastructures of *Standard ML* and the *JVM*, *Scala* was chosen as the *JVM* communication endpoint because, due to its flexibility and functional features, it allows programmers to imitate a programming style similar to *Isabelle/ML*. *Scala*'s actors (section 3.4.2) were used heavily for asynchronous communication and concurrency until recently. Since the original Actor library was replaced by *Akka*, *Isabelle* developers have decided to imitate the *ML* variants of parallelism mechanisms in *Scala* without actors [119]. All datastructures that are part of the protocol have been implemented in both languages side by side. E.g. the datatype definition of file system path elements is

```
datatype elem =
  Root of string |
  Basic of string |
  Variable of string |
  Parent;
```

in *Standard ML*[4]. Here, a value of type `elem` can be either of the four subtypes, three of which are `string`s. The equivalent datastructure in *Scala*[5] is given as four concrete classes inheriting from the abstract class `Elem`:

[4] `~~/src/Pure/General/path.ML`
[5] `~~/src/Pure/General/path.scala`

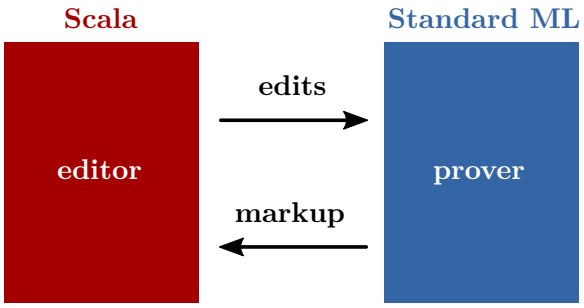

Figure 4.2: The *Isabelle/PIDE* communication model [93].

```
sealed abstract class Elem
private case class Root(val name: String) extends Elem
private case class Basic(val name: String) extends Elem
private case class Variable(val name: String) extends Elem
private case object Parent extends Elem
```

The internal protocol utilizes *YXML*, a transfer format based on XML [94]. Instead of keeping a public protocol specification, the details are being kept private. This allows the developers to make substantial changes to protocol and document model to improve robustness and performance without further ado. Instead, both *Isabelle/ML* and *Isabelle/Scala* offer public, static APIs for their communication which are maintained and kept simple and stable.

Fig. 4.2 outlines the basic communication model between the *JVM* and the prover. Whenever the user edits a file in an editor, a description of this update is sent to the prover which then produces markup to annotate the new contents with proof data and other semantic information. Then the changes on the editor side are reverted and replaced by the new, annotated content. As a consequence, the interaction mechanism adheres to the functional paradigm's philosophy by using strictly immutable document snapshots of versioned file histories.

4.1.4 *Isabelle/jEdit*

Recent *Isabelle* distributions include a complete prover IDE prototype implementation based on the text editor framework *jEdit* [95]. *Isabelle/jEdit* features a completely new interaction model that performs continuous, parallel proof checking. It uses GUI functionality such as highlighting, coloring, tooltips, popup windows and hyperlinks to annotate user code written in *Isar* and *ML* incrementally with semantic information from the prover (see fig. 4.3). Prover and editor front-end are executed independently and never block each other thanks to the asynchronous protocol. Additionally,

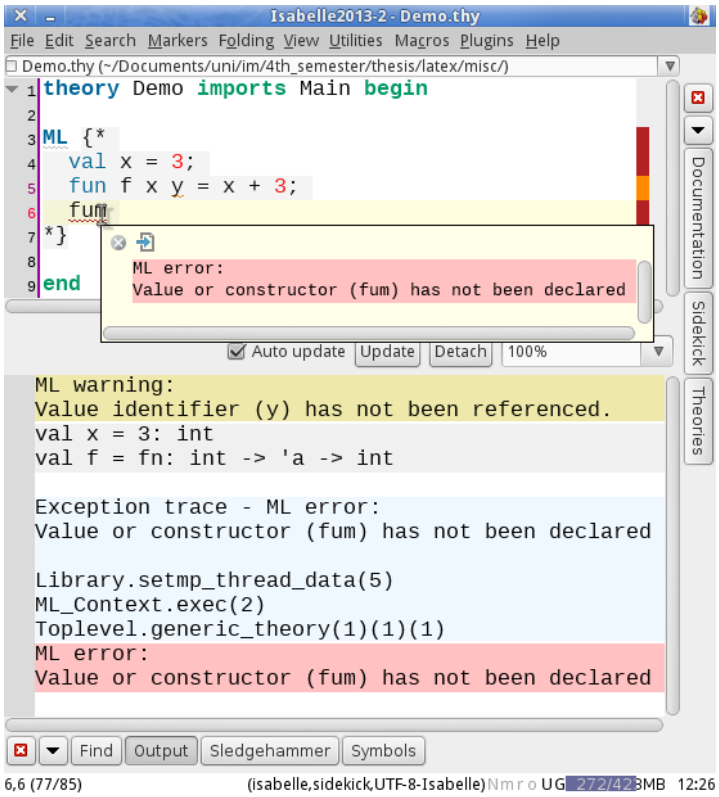

Figure 4.3: GUI feedback in *Isabelle/jEdit*.

Isabelle/jEdit includes several plugins and fonts to facilitate entering and rendering of mathematical symbols.

The IDE also manages loaded theory files and the computation of their dependency graph. Changes to theories are automatically propagated appropriately and the user receives immediate feedback on the processing of theories (fig. 4.4).

4.1.5 The Prover Framework

The paragraph titled *The Next Seven Hundred Theorem Provers* in section 2.2.1 (page 8) explained that *Isabelle* was initiated and designed as a logical framework for automated theorem provers. Its design is described in [97]. The metaprover / collection of external automated provers supporting interactive proofs is called *Sledgehammer*. The standard provers are called *E*, *SPASS*, *Vampire* and *Z3*. Fig. 4.5 (page 38) depicts the view on *Sledgehammer* in *Isabelle/jEdit*. In those cases where theorems are invalid and therefore cannot be proved, two disproof tools called *Quickcheck* and *Nitpick* can be utilized for producing counterexamples. See [11] for a more detailed overview

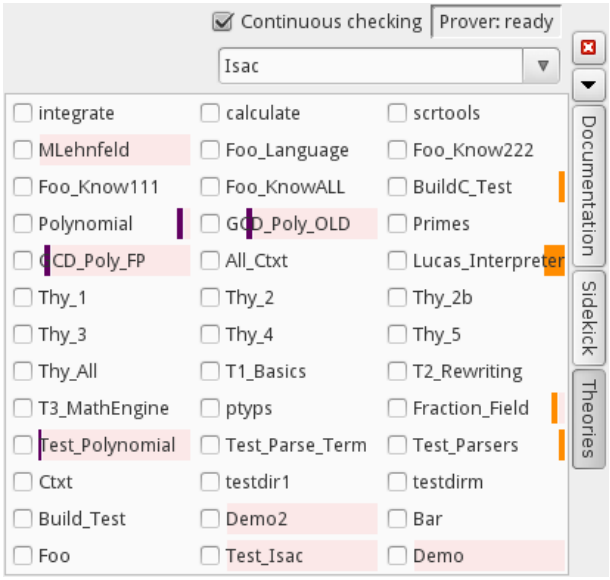

Figure 4.4: Feedback on theory processing *Isabelle/jEdit*.

on the integration of these external tools.

4.1.6 Development and Documentation

Apart from the technologies used in *Isabelle*, also *Isabelle*'s development process and system documentation had an impact on this thesis. The adaption to *Isabelle*'s specifics was quite challenging. *Isabelle*'s development process is much faster and much more radical as compared to commercial software development due to the specific kind of development. This development is led by three persons (see introduction to section 4.1, two of them supervising and also writing code themselves) and executed by a small team of about ten postdocs and PhDs, each working under contracts for three years at least. The members grow into the team by doing projects and master theses before entering the team. According to my supervisor, this particular kind of development leads to the following way of system documentation which is given by:

1. **The code** itself, without any further comments except in few specific cases. This is common to functional programming and very different from what one is used to from e.g. *Javadoc*. The code can be inspected in the repository, see pt. 3 below.
2. **Reference manuals and tutorials**, which can be found online [102] and are easily accessible from *Isabelle/jEdit* (section 4.1.4). Both, manuals and tutorials are automatically held in sync with the code by specifically developed mechanisms such as antiquotations (section

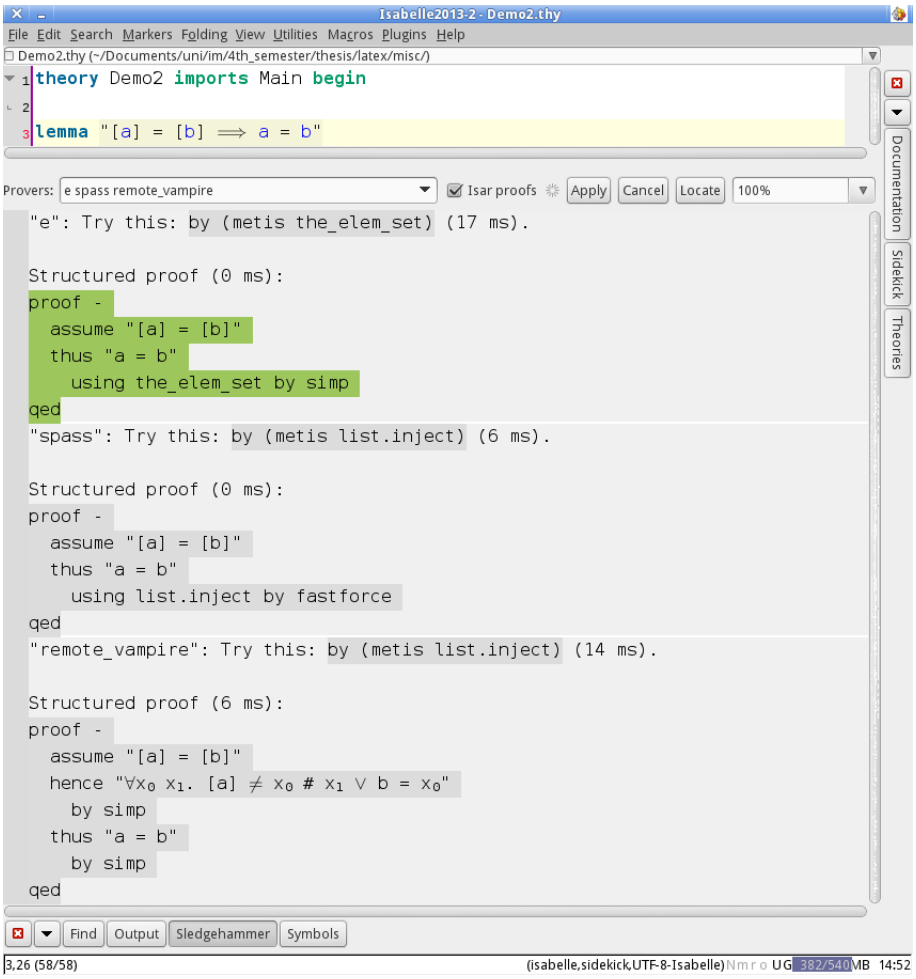

Figure 4.5: instance of the *Sledgehammer* panel.

4.1.1, page 30), e.g. `@{type ...}`, `@{const ...}`, etc. These entries in the documentation raise exceptions in case some type or constant has changed in the code.

3. **The repository**, publicly readable [110] is the third pillar for documentation. Commits must follow a *minimal changeset* policy which allows for quick queries and documents particular interdependencies across several files. This kind of documentation of complicated connections is very efficient and always up to date even with frequent changes.

This documentation model has evolved during more than two decades and still allows for ongoing rapid development with invasive changes, last but not least parallelization. However, relevant documentation for this thesis is marginal. Parallelization, as already mentioned, has been done by Makarius

Wenzel and was preceded by a feasibility study conducted by a master student [34]. Wenzel's respective papers [60, 92, 94–96] concern fundamental considerations. The most relevant reference manual [98] contains less than ten pages on parallelization. All the rest had to be found in the code (several hundred thousand LOCs[6]). The gap between papers and code was hard to bridge.

4.1.7 Reasons for Parallelism in *Isabelle*

Isabelle is the first theorem prover which is able to exploit multi-core hardware. The reasons for introducing parallelism in *Isabelle* are directed towards the future and complement considerations discussed for functional programming. There are three motivations for parallelization mentioned for *Isabelle*. They have all been addressed at once: parallel invocation of multiple automated provers, a responsive prover IDE and efficient theory evaluation envisage better usability in engineering practice. All three development issues are successfully being tackled, an end cannot be foreseen yet.

Run Automated Provers in Parallel

Isabelle's collection of automated provers was mentioned in section 4.1.5. It still has a mechanism to connect with remote prover instances and these already featured parallel access to external resources. Since personal computers and laptops have become powerful enough to run *Isabelle*, this hardware has also become powerful enough to run automated provers. But running them in parallel on a local machine required a redesign.

Provide a Tool Usable in Engineering Practice

The paragraph about *Continuous Penetration of Computer Science* by formal methods in section 2.2.1 (page 10) mentioned the increasing demand for theorem provers in the practice of software engineering and related mathematics. In particular, software engineers are used to comprehensive support by integrated development environments (IDEs): interlinked code, various views on relations between code elements, etc. Since TPs have been used by experts only until the presence, specific user interfaces were sufficient. For *Isabelle* this was *Proof General* (see section 4.1.4), which is based on *Emacs*. Those who still use this editor which was dominant during the last decades, know how different *Emacs* is: not reasonable for engineering practice of today. So the issue was to meet the usability requirements of contemporary engineering practice. *Isabelle*'s attempt to meet such requirements is the *PIDE* project (section 4.1.3). *Isabelle/jEdit* (section 4.1.4) is the reference

[6] *lines of code*

implementation, which is accompanied by development of an *Eclipse* plugin [111] and a browser-based front-end under construction.

Accelerate theory evaluation

My supervisor's first experience with *Isabelle* was about fifteen years ago on a high-end SPARK workstation. The initial evaluation of *Isabelle/HOL* took more than twenty minutes; not to speak of the considerable amount of time an experienced system administrator needed for installation. Since this time *Isabelle/HOL* has become about three times larger, but the initial evaluation of theories takes about two minutes for recent *Isabelle* releases. *Isabelle/jEdit* represents theory evaluation as shown in fig. 4.4 on page 37. Parallel branches in the dependency graph are evaluated in parallel. Without this kind of speedup, large theory developments involving several dozens of theories would not be manageable. The most recent *Isabelle* releases do not package binaries of evaluated theories any more. Instead, they pack further components into the bundle. The bundle of the current release is more than 400MB for download. Also, all other bandwidth aspects of *Isabelle* are at the limits of present mainstream hardware. *Isabelle* does not run below 4GB main memory and requires a dual-core processor with more than 2GHz per core.

4.2 *Isac*

Isac is a prototype in both, in the mathematics-engine based on *Isabelle* and written in *Standard ML* as well as in the front-end involving rule-based dialogs and written in *Java*. The mathematics-engine heavily reuses technology from *Isabelle* and thus comprises a comparably small portion of code: 25,000 LOCs[7] plus 15,000 LOCs of knowledge in theories compiled to *Standard ML* for fast access and 40,000 LOCs for tests. This code features functionality which is radically new for educational mathematics systems. The gain from TP technology is explained in section 4.2.1. The front-end exploits the mathematics-engine's TP-based functionality for dialogs on a new level of flexibility and user guidance. The gain from rule-based dialog guidance is addressed in section 4.2.2. Since *Isabelle* is still under rapid development, the principal question arises, whether *Isac*'s prototype development can persistently adjust to *Isabelle*, an issue addressed in section 4.2.3. Otherwise the success achieved with the case study on parallelization would be worthless in the long term. The description of the case study is comprised of two parts: the integration with *Isabelle*'s parallel theory evaluation is described in section 4.3.1 and section 4.3.2 discusses the parallelization of *Isac*'s user session management.

[7] *lines of code*

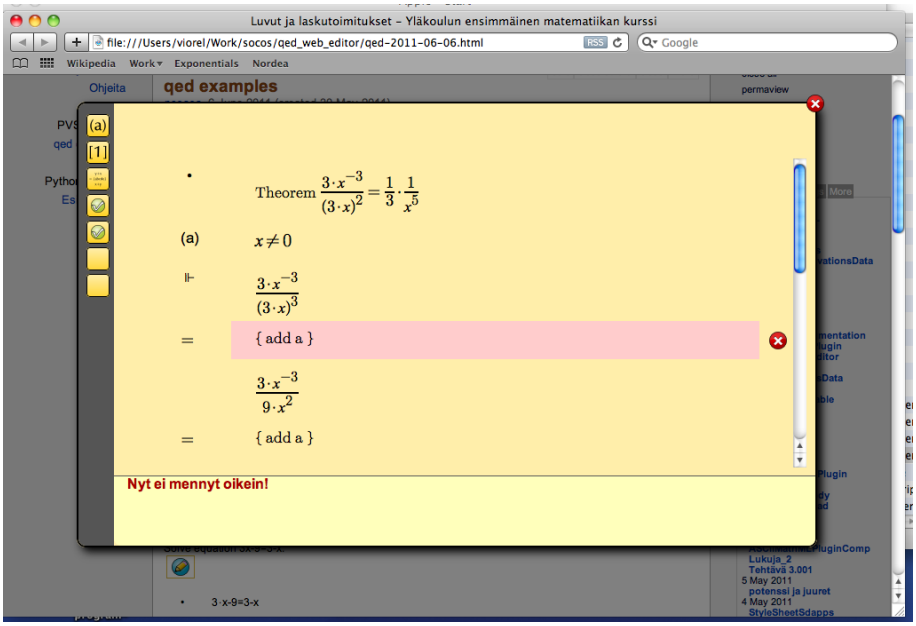

Figure 4.6: Structured derivation editor (source: [7]).

4.2.1 Principal Design Decisions

Isac is still the only educational mathematics system worldwide which adheres to concepts of TP and which is built reusing TP technology, except for a system under construction by the Scandinavian *E-Math* project [103]. All the other math tutoring systems are based on computer algebra. Thus, the distinctive design decision to rely on TP deserves an explanation.

TP concepts ensure logical correctness in stepwise problem solving. When learning independently from a human tutor or supervisor, a student wants to know whether the calculations are done correctly or not. An educational software system supporting this expectation cannot be based on computer algebra systems. These are in no way related to logics (which is an unresolved issue for computer algebra, for instance at *RISC Linz*), so true / false or correct / not correct requires underpinning with logical concepts as is the case e.g. in the *Theorema* project at *RISC Linz*. The system under construction in *E-Math* supports so called *structured derivations* which are close to calculations written by hand (see fig. 4.6).
This kind of calculation is proved logically equivalent [7] to natural deduction, which forms the logical foundation of *Isabelle* and other provers as mentioned in section 2.2.1.

With the decision for *Isabelle*, *Isac* intentionally dissociates itself from the kind of artificial intelligence that tries to imitate human behavior. *Isac*

does not intend to mimic a human tutor by modeling the system with respect to human thought, human comprehension or human learning. Rather, *Isac* is designed as a model of mathematics, following the structure of mathematics from the very bottom, from formal logics as implemented in *Isabelle*. However, AI technologies can take profit from the strength, the generality and the flexibility of the mathematics-engine resulting from this design. The principal strengths of *Isac*'s math engine are described in the following paragraphs.

TP technology features new functionality of *Isac*'s mathematics-engine. The following features arise naturally from TP:

Check students' input automatically, flexibly and reliably. Given a problem of applied mathematics by a formal specification (input, output, precondition and postcondition; generally hidden from the student), any formula (or a theorem for application) input by the student establishes a proof situation. Automated provers then derive, if possible, the input from the logical context. This is what computer theorem provers are built for. They represent the technology for accomplishing this task as generally and reliably as possible at the present state of the art in computer mathematics.

Give explanations on request by students. Provers designed on the basis of the *LCF principle* (section 2.2.1) such as *Isabelle*, derive all knowledge from "first principles". For each definition and for each theorem the respective prerequisites can be traced down to the most fundamental axioms. All this knowledge is represented in human readable form [112]. Thus the situation in systems like *Isabelle* is quite different from computer algebra systems. While documentation for the latter requires additional efforts for creation, all documentation is already there in *Isabelle*. Generation of explanations calls for filtering out details, which could distract or overwhelm students. Another nice feature of *Isabelle/Isar*, although not relevant for applied mathematics, are human readable proofs close to mathematical traditions (see section 4.1.2 and appendix A.3).

Propose a next step if students get stuck. This feature seems indispensable for independent learning. However, this exceeds the traditional discipline of TP. E.g., *lemma* $\int 1 + x + x^2\ dx = x + \frac{x^2}{2} + \frac{x^3}{3} + c$ can be proved, but given $\int 1 + x + x^2\ dx$ there is no direct way to calculate the result $x + \frac{x^2}{2} + \frac{x^3}{3} + c$ in TP. Since indispensable, *Isac* provides this feature by a novel combination of deduction and computation, called *Lucas-Interpretation* [67]. *Lucas-Interpretation* works on programs which describe how to solve a problem of applied mathematics in a quite traditional way. In addition to maintaining an environment as usual for interpretation, *Lucas-Interpretation* maintains

a logical context, such that automated provers are provided with all current knowledge required to check user input. [22] gives a brief introduction to *Lucas-Interpretation*.

These three features together establish a powerful interface to the mathematics-engine. With these given, interaction can be analogous to learning how to play chess in interaction with a chess program. The player executes a move (the student inputs a formula) and the program checks the correctness of the move (*Isac* checks the input). The program performs a move (*Isac* proposes a next step) and the player accepts or changes it. If the player is in danger of losing the game (the student gets stuck within the steps towards a problem solution), they can go back a few moves and try alternative moves. Or the player can even switch roles with the system and watch how the system copes with the unfortunate configuration. All these choices are available to a student analogously when learning mathematics in interaction with *Isac*.

The comparison of interaction in *Isac* with interaction in playing chess makes clear, that learning with a mechanical system does not necessarily lead to drill and practice in operational mechanization. The degree of freedom in interaction allows for fostering of the evaluation of alternatives and thus learning by trial and error, comparing strategies, etc.

4.2.2 Architecture Separating Mathematics and Dialogs

The strengths of *Isac*'s mathematics-engine, as described above, provide powerful features for human-machine interaction and thus call for concepts and technology from AI. In order to cope with increasing complexity, a separation of concerns between AI and mathematics is required.

Separation of Concerns between Dialogs and Mathematics

This separation is reflected in *Isac*'s architecture shown in fig. 4.7. The front-end on the left within a *Java* environment comprises *GUI 1..n*, an arbitrary number of remote devices for input and output, the *Isac server* with the *dialog* component. The *dialog* component comprises a rule-based system as described in [50], not shown in the figure. *Dialog authors* are not concerned with mathematics and are free to focus on the complexity of dialog guidance, see [22] and an example from the paper in fig. 4.8. In fig. 4.7, the server accesses the *knowledge* in HTML representation, both in turn administered by a learning management system like *moodle*. Separated from the front-end concerned with dialogs etc. is the *Standard ML* environment, the implementation language of *Isabelle* and *Isac*: the *math-engine* together with *programs* for *Lucas-Interpretation* and *knowledge* in *Standard ML* representation for fast access. The math-engine builds upon *Isabelle* by reusing the term structure of simply typed lambda calculus and respective parsing, pretty printing,

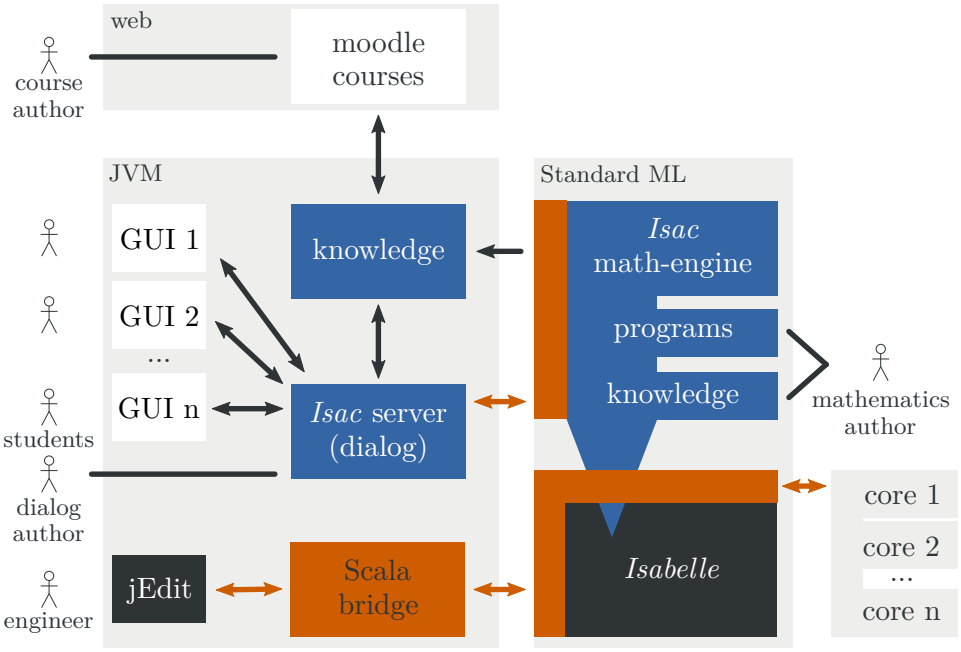

Figure 4.7: *Isac*'s architecture as seen by users.

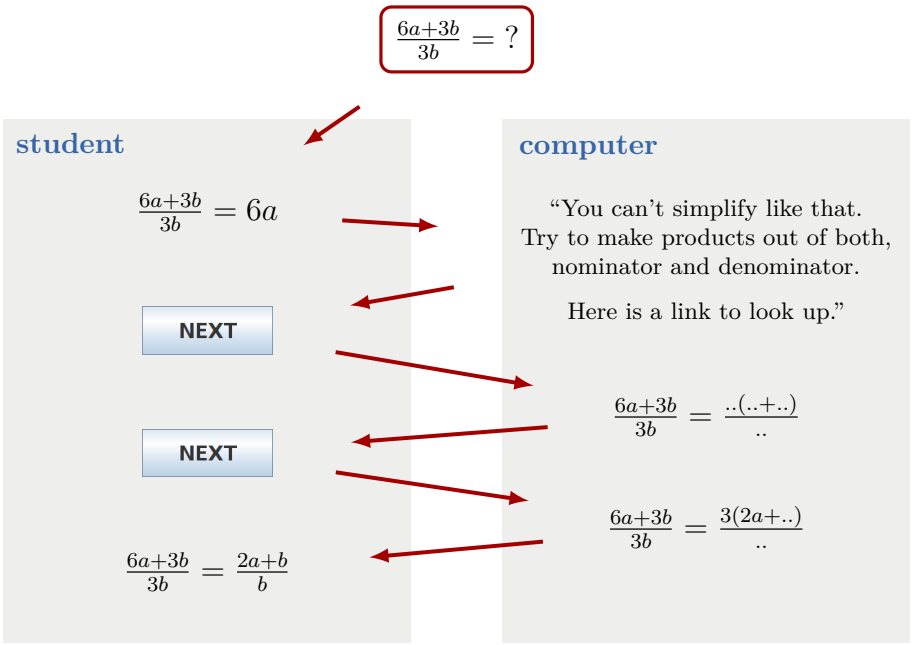

Figure 4.8: Interaction with *Isac* [22].

matching as well as management of *knowledge* within theories. The orange color in fig. 4.7 indicates parallelization in order to exploit availability of an arbitrary number of cores (*core 1..n*). In 2009, *Isabelle* introduced specific mechanisms to cope with multi-core environments. The central *Isabelle* component for parallelization is the *Scala bridge* (section 4.1.3) supporting asynchronous user interaction [92] and parallel proof checking [97] via the editor *jEdit* (section 4.1.4). *Isabelle*'s parallel mechanisms are reused in the case study. The horizontal orange bar above *Isabelle* indicates that parallelization of *Isabelle* affected *Isac*'s management of *knowledge* in *Standard ML* representation. This part of the case study is described in section 4.3.1. The vertical orange bar to the left of the *Isac math-engine* indicates the concurrent user session management accomplished by the second part of the case study, which is described in section 4.3.2.

Modeling the Universe of Mathematics

This goes beyond *Isabelle*'s way of modeling the deductive dimension of knowledge. Due to a suggestion by Bruno Buchberger in 2001, *Isac* now implements two additional dimensions, together spanning a three-dimensional space:

1. **Deductive knowledge** is defined and managed in TP. Beginning with the basic axioms (axioms of higher-order logic in *Isabelle/HOL* used by *Isac*), definitions are given and theorems about these definitions are proved, theory by theory. *Isabelle* provides an elegant mechanism for defining new syntax, which results in a language close to traditional mathematical notation. The subset of *Isabelle* knowledge presently in use by *Isac* can be found online [123].
2. **Application-oriented knowledge** is given by formal specifications of problems in applied mathematics. The specification of all problems, which can be solved automatically by *Isac* (with the help of algorithms, see below), is available online [121]. This collection is structured as a tree. Given a tree, automated problem refinement is possible, e.g. starting from the specification at the node of *univariate equations* [122] a certain equation can be matched with all child nodes until the appropriate type of equation has been identified. In case a match has been found, such a node contains a pointer to a program solving the specific type of equation. This kind of problem refinement makes transparent, what is done by computer algebra systems as black boxes.
3. **Algorithmic knowledge** is given by programs to undergo *Lucas-Interpretation*. The collection of programs is structured as a tree. However, principal requirements on the structure are still unclear. There are ideas of how to group algorithms in [16] but the ideas are too vague for mechanization. *Isac*'s programs can be found online [120].

The structure of the knowledge collections of pt. 2 and 3 are different from pt. 1. E.g. linear equations in pt. 2 are the same for rational numbers and complex numbers, but the definition of rationals is far away from complex numbers in pt. 1. So the implementation of the latter two collections is orthogonal to the first, the collection structured and managed by *Isabelle*. The only way of implementation was by `Unsynchronized.ref`, i.e. by global references. Write access to these references assumed a certain sequence in evaluation of the theories. This assumption broke with the introduction of parallel execution in *Isabelle*. Section 4.3.1 explains how this issue was resolved.

4.2.3 Relation to Ongoing *Isabelle* Development

In my Bachelor's project [54] I already had the chance to contribute to *Isac*. From my experience with *Isac* I can give the following overview of *Isac*'s development.

Although being active for more than two decades, *Isabelle* development still undergoes invasive reforms concerning localization, parallelization, etc. See the NEWS file [113] for details. These reforms cause significant changes in module signatures and in functionality more or less with each release. Many of these changes enforce considerable updates of *Isac* because *Isac* extracts functionality from *Isabelle* at a rather deep level, apart from public interfaces. Updating *Isac* in accordance to the regular *Isabelle* releases is crucial: Between 2002 and 2010 *Isac* development focused on the front-end. In 2010, *Lucas-Interpretation* [67] was clarified and *Isabelle*'s contexts appeared indispensable. Contexts appeared in *Isabelle* after 2002, so an update was necessary. However, updating *Isac*'s math-engine from *Isabelle2002* to *Isabelle2009-2* turned out to be very hard and it was done during and after my Bachelor's project. Ever since this unpleasant experience *Isac* has been updated with each official *Isabelle* release which are published every eight to twelve months. In addition to these regular updates there are major tasks concerning further adoption of *Isabelle*'s mechanisms which are open due to early design decisions of *Isac*.

Adopt logical contexts from *Isabelle*. *Isac*'s mathematics-engine was developed between 2000 and 2002, long before *Isabelle* introduced logical contexts [98]. Without these contexts the user needed to explicitly fix the type of variables during input. This was very inconvenient. I implemented my Bachelor's project in time to introduce contexts to *Isac* and gain first experiences with the system. Experience with contexts also contributed to the theoretical clarification of *Lucas-Interpretation* [67].

Adapt to *Isabelle*'s parallel theory evaluation. Since *Isabelle* traverses its theory dependency graph in parallel, *Isac* can no longer rely on a

particular, deterministic execution order. Fixing the problems which arose from this fact and integrating the related datastructures into *Isabelle*'s theory data management model is one of the big achievements of the project work that this thesis is based on. The details of this process are outlined in section 4.3.1.

Exploit parallelism for concurrent session management. This task addresses the utilization of *Isabelle*'s new mechanisms for parallelism, most notably futures, for *Isac*'s user session management in order to allow for a high number of concurrent, efficiently executed calculations. Section 4.3.2 documents this step in more depth.

Make *Isac*'s programming language usable. *Isac*'s programming language was implemented between 2000 and 2002, long before *Isabelle*'s function package was available [51]. *Lucas-Interpretation* on programs in this language uses a variety of rulesets for rewriting, too complicated for an average programmer [79]. Considerable simplification of programming can be expected by narrowing *Isac*'s programming language towards *Isabelle*'s function package. In particular, components of *Isabelle*'s code generator [32] promise to greatly improve computational efficiency.

Adopt *Isabelle*'s numeral computation for *Isac*. The initial conception of *Isac* emphasized applied mathematics, which is unrealistic without floating point numbers. There were early attempts to implement the latter together with complex numbers. Both, floating point numbers and complex numbers have in the meantime been implemented in *Isabelle*. These shall now be adopted for *Isac*.

Improve the efficiency of *Isac*'s rewrite engine. *Isac*'s initial design stressed the importance of computations closely resembling work by paper and pencil. Major parts of computation happen by rewriting, so a large number of rewrite rules need to form groups which as such mimic the step width of human computations. *Isabelle*'s rewrite engine meets opposite requirements: provide a maximum of automation with large rule sets ("simpsets"). Only recently, frequent user requests motivated projects in the *Isabelle* development to provide readable traces for rewriting. However, the requirements of *Isabelle* and *Isac* are too different and *Isac* will keep its own rewrite engine. But the mechanism of *term nets* [18] could be adopted to improve *Isac*'s efficiency.

Adopt *Isabelle/jEdit* for *Isac*. *Isac*'s initial design also stressed usability of the front-end for students. At this time *Isabelle*'s front-end was *Proof General* [6]. Thus there was no choice but to develop a custom GUI for *Isac*.

In the meantime *Isabelle/jEdit* has become an appealing prover IDE. However, the requirements are presently quite opposing. *Isabelle* connects one engineer with a multitude of cores while *Isac* connects a multitude of students with one server. *Isac*'s centralized structure has good reasons: groups of students have one lecturer, groups of students are examined together, etc. During the next years, *Isac*'s front-end will be developed and improved independently from *Isabelle*. Narrowing towards *Isabelle* can start as soon as *Isabelle/jEdit* moves towards collaborative work, implements session management, version management, etc.

Summary on *Isac*'s Approach to *Isabelle*

Above, those points are listed, where *Isac*'s math-engine is supposed to approach *Isabelle* in future development, as this thesis already has begun. Development of *Isac*'s math-engine is separated from the development of *Isac*'s front-end. Section 4.2.2 describes *Isac*'s *architecture* separating *mathematics and dialogs*. Development of the front-end addresses the expertise represented by our *University of Applied Sciences* in Hagenberg: human-centered computing, interactive media, mobile computing, secure information systems, software engineering, etc. *Isac*'s development efforts in these disciplines can be planned independently from the above list for several years, until *Isabelle/jEdit* is ready to be adopted for *Isac* in a particularly interesting development effort.

4.3 Practical Work on *Isac*

This section documents the implementation work that has been carried out as project work in preparation for and along with the writing of this thesis. During the first big step, the breakdown of *Isac*'s theory evaluation caused by an invasive update to *Isabelle*'s mechanisms was fixed (section 4.3.1). This made possible the introduction of concurrency to *Isac*'s user session management and thus enabled the parallel computation of independent calculations. This process is described in section 4.3.2. Section 4.3.3 discusses the effects of this process on *Isac*'s performance. Finally, *Isac*'s project status is summarized in section 4.3.4.

4.3.1 Integration With *Isabelle*'s Parallel Theory Evaluation

In older *Isabelle* versions, it was not possible for programmers to append arbitrary data to the datastructure which holds all the theories. *Isabelle2005* [113] then introduced a concept called *theory data*, which allowed developers to store user-defined datastructures in a theory's context from *ML* code embedded in that theory.

Data flow. In order to understand the following paragraphs, we need to know how *Isabelle* computes theories and what the theory data flow looks like. Section 4.1.2 already mentioned, that the inheritance structure of *Isabelle*'s theories forms an acyclic, directed graph. Computation begins at the root theories which are determined backwards from the current working theory. The resulting data is then available in all descendant theories. Whenever a theory has more than one ancestor, the derived data is merged.

The last sections have outlined the conceptual and architectural differences between *Isabelle* and *Isac*. *Isac* needs to manage application-oriented and algorithmic knowledge which originally could not be integrated with the deductive structure of *Isabelle*'s theories. Therefore these data were stored in raw ML references. *Isabelle2009* then introduced parallelism. Now the computation order of the theories was not deterministic anymore and *Isac* could not ensure that its *ML* references were accessed and updated in the same order as previously. This caused certain parts of the system to show faulty behavior in some cases. *Isabelle2009-1* then added the wrapper structure `Unsynchronized.ref` to denote that these references are not synchronized with the parallel environment managed by *Isabelle*. While the temporary deactivation of faulty modules and certain workarounds fixed most problems, the parallelization of *Isac*'s user session management required that most relevant data be properly managed by *Isabelle*. Therefore the preparatory step for the parallelization was the integration of the unsynchronized references in question with *Isabelle* by means of the functor `Theory_Data` [98]. It is important to mention that the *Isabelle* implementation manual [98] warns to be careful about using too many datastructures on the basis of `Theory_Data` because they are newly initialized for every single theory that derives from the respective module that declared the datastructures. Thus, space is reserved which can cause a significantly increased memory footprint. Most of the overhead, however, occurs when theories are loaded dynamically. When working with a compiled *Isac* system the overhead should be reasonable.

The workflow was implemented for several central elements and broken down into five isolated steps to conform with *Isabelle*'s minimal changeset development and documentation model (section 4.1.6):

1. **Isolate access.** In this step, read accesses were centralized by wrapping references in an access function and replacing all occurrences with a call to the function:

```
val foo = Unsynchronized.ref [1,2,3]; (*declaration*)
fun get_foo () = !foo; (*new*)
fun add x y = x + y;
fun foo_sum () = fold add (!foo) 0; (*obsolete*)
fun foo_sum () = fold add (get_foo ()) 0; (*new*)
```

2. **Add `Theory_Data` access functions** in parallel to the existing ones. This includes a new version of the getter function (`get_foo` in pt. 1). But first we need to define a datastructure that *Isabelle* can manage. Thus, we need to implement the functor `Theory_Data` for our specific datatype[8]:

```
structure Foo_Data = Theory_Data (
  type T = int list;
  val empty = [];
  val extend = I; (*identity*)
  val merge = merge_lists;
);
```

`T` is the type of the data. `empty` is the initial value for every theory that does not define this particular data slot content. `extend` is used for reinitialization on import of a theory and can be understood as a unitary `merge`. Finally, `merge` declares, how the data slot contents of two theories are to be joined. Unlike the operations that we are replacing during this process, the two simple access functions

```
val get_foo' = Foo_Data.get;
fun put_foo' xs = Foo_Data.map (merge_lists' xs);
```

additionally require the target theory as input parameter. Because *ML* code is always embedded in *Isabelle* theories, they can operate on any available theory according to the dependency graph. Each theory stores its own version of the data slot. A new version of `get_foo`,

```
fun get_foo () = get_foo' @{theory}; ,
```

which is not yet in use during this phase, could replace `get_foo` from pt. 1. This definition uses antiquotations (section 4.1.1, page 30). This means that `@{theory}` always refers to the theory where the function definition is located and is thus resolved at compile time. The alternative approach

```
fun get_foo () = get_ref_thy () |> get_foo';
```

is more flexible in that `get_foo'` operates on a reference theory set by the programmer (details on `get_ref_thy` are explained in the subsequent section) and passes it to `get_foo'` from above. The last aspect of this step is the addition of **setup** blocks [99] where previously the raw references had been updated. These blocks must contain one *ML* expression which represents a mapping between two `theory`s. As a consequence, *Isabelle* updates the current theory context with the function's result in these locations, e.g.

[8] see appendix A.2 for definitions of `merge_lists(')`

```
setup {* put_foo' [4,5,6] *} .
```

3. **Check differences.** Now that both, the old unsynchronized reference and the new stored as theory data, are created and updated, we can compare their contents and make sure that the last steps were successful. Due to the nature of the datastructures and the fact that the new method caused a different ordering to some of the elements, the most convenient solution was to compute a string representation from both results, write them into documents and compare the documents using a file comparison utility. If any errors occurred, step 1 and 2 required revision.
4. **Shift to theory data.** Optimally, this step merely meant exchanging the old definition of `get_foo` (pt. 1) for the new one (pt. 2).
5. **Cleanup.** Finally, we can remove all code concerned with the unsynchronized reference.

Besides acquiring the necessary knowledge on how to store and access arbitrary datastructures in a theory's context, the biggest challenges included understanding and merging the different kinds of custom datastructures and keeping the solutions simple; optimally simpler than the previous code. Also, I had to adjust the theory hierarchy and add new theories in order to keep a clean structure and ensure that the availability and computation of datastructures was sound and behaved as it had previously. Some of the data dependencies had not been reflected in the dependency graph but had rather been fulfilled by a fortunate execution order.

As a result of this contribution, most of the stateful components in *Isac*, which had been necessary to circumvent mismatches between *Isac*'s and *Isabelle*'s architectures, were eradicated from the system. Those which are still not synchronized are currently being replaced. However, they are not accessed by theories potentially executed in parallel and therefore do not interfere with *Isac*'s parallel user session management.

4.3.2 Introduction of Concurrent User Session Management

Isabelle's programming conventions prohibit user programs based on *Isabelle* from forking their own *ML* threads. This is necessary because *Isabelle* has its own managed environment for concurrency which takes care of thread synchronization and shared resources. Accesses from user threads could easily break this well-tested framework. After careful analysis of the *Isabelle* documentation and the code in some concurrency related *Isabelle/ML* modules, our suspicion that parallelizing *Isac*'s user session management required breaking this convention was rendered unfounded. As it turned out, *Isabelle*'s implementation of *future*s (section 4.1.1) provided everything we needed. As a result, the actual implementation effort was marginal. Since *Isac*'s does not maintain any problematic stateful resources anymore, there was no further

effort required for synchronizing shared resources beyond those managed by *Isabelle*, except for the `states` element, which maintains a state for each active calculation. Instead of an `Unsynchronized.ref`, it was replaced by a `Synchronized.var` (see section 4.1.1, page 30).

The function `appendFormula`, which derives the input formula from a calculation's state, had to be transformed in a way, such that it can be passed to `Future.fork` which then takes care of the parallel execution.

```
fun appendFormula params =
      Future.fork (fn () => appendFormula' params);
```

is a simplified version of the key line in this transition, with `appendFormula'` being the original version of the function. The same transformation was applied to `autoCalculate`, which can compute calculations step by step as well as automatically determine the final result. Many of the computations within `autoCalculate` require access to the current proof context datastructure. During the previous part of the project (section 4.3.1), this was done using the function `the_generic_context` from *Isabelle*'s *ML_Context* module. When I called more than one instance of `autoCalculate` simultaneously, it turned out that functions executed within a future cannot acquire access to the current proof context through this function. In practice, the context data of the theory *Isac* is sufficient for all calculations and could be completed and made available at compile time. However, the *Isac* test suite extends the context data, which cannot be modified at run time. In order to overcome this, there is now a `Synchronized.var` at an appropriate location within the dependency graph, which determines the reference theory for all operations accessing the newly introduced `Theory_Data` elements. The test suite uses a new function `set_ref_thy` to override the standard value which is the theory *Isac*.

Now that all calls to `the_generic_context` have been eliminated, invocations of `autoCalculate` and `appendFormula` return immediately with a future value and thus the respective operations are executed in the background.

4.3.3 Performance

The main objective of my project was to improve the speed with which calculations are performed by the system when multiple user front-ends invoke the *Isac* mathematics-engine simultaneously.

The change in memory usage is not our foremost concern and it can only increase linearly. The number of concurrent calculations depends on *Isabelle*'s thread pool size. Pending invocations of the mathematics-engine are problem descriptions whose memory footprint is negligible, considering a reasonable number of clients working on the same *Isac* server. One concern

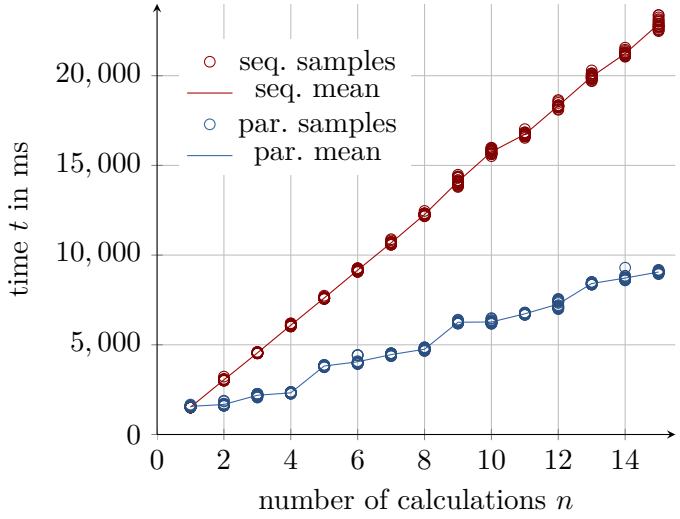

Figure 4.9: `autoCalculate` performance comparison.

is the keeping of state data on a per user basis which could accumulate and pose a challenge to the server's working memory under heavy load.

Temporal gains were measured on a 64-bit *Arch Linux* system running on a *Lenovo ThinkPad X230* with 4GB RAM and an *Intel® Core™ i5-3320M CPU* with two physical 2.60GHz cores and one additional logical core respectively (see *simultaneous multithreading* in section 2.3.2). The performance test file can be found on the attached DVD (see appendix B.2). It uses the most CPU-intensive problem instance currently available, whose calculation takes about 1.58 seconds on average. Various numbers of parallel invocations were issued and the time until all of them were finished was measured. Per test there were 20 repetitions and an average taken. All the samples are shown in fig. 4.9.

Discussion

From these results it is evident, that two parallel calculations can almost fully exploit both physical processing cores. The slight increase of processing time can be explained by a parallelization overhead caused by *Isabelle*'s future management. Another factor are the synchronized accesses to the `states` datastructure. Since simultaneous multithreading cannot completely simulate additional cores, the increase in runtime between two and three parallel calculations is then slightly bigger. The same happens between six and seven parallel `autoCalculate` invocations. As the number of logical cores of the used machine is four, multiples of four show a significant increase in runtime because once all cores are occupied, additional calculations need to wait. Hence, the execution time accurately doubles with the number of

calculations according to the number of cores. The measured runtime of the new `autoCalculate` version was only 38.7% of the previous, sequential implementation with eight parallel calculations, 39.8% with ten and 39.7% with twelve. This is a very satisfying outcome and we trust that it will help to significantly improve the end user experience and enable *Isac* servers to deal with a high number of clients simultaneously.

4.3.4 Project Status

The user session management is now able to utilize several processing cores for independent computations. However, communication between the *ML* and *Java* layers occurs on the basis of plain standard I/O streams, i.e. multiple front-ends using the same math-engine instance need to write to the same input stream on the *ML* side and thus communication is unsynchronized. This means that although the math-engine is now able to deal with calculations for multiple users concurrently in an efficient manner, the communication model does not yet allow for thread-safe communication. Changing this may be subject for a future project. Another area requiring further investigation is the memory footprint of accumulated state data for calculations and how to deal with a very high number of concurrent user sessions in this respect.

Chapter 5

Conclusion

In this thesis we showed how an educational mathematics system called *Isac* was integrated with the theory data evaluation scheme managed by *Isabelle*, an interactive theorem proving system. This modification allowed for the straight forward introduction of a parallel execution mechanism called *futures*, which are data structures holding possibly unfinished computations and whose evaluation is efficiently managed by *Isabelle*'s run-time system. Before we went into the details of this process, a comprehensive theoretical basis was established. This included a discussion of the requirement for modern software to utilize the processing power available on shared memory multi-core architectures which have become a standard during the last decade. Also, various approaches towards the implementation and benefits of parallelism and concurrency in functional programming were explored, along with important aspects to consider in order to parallelize software efficiently. The architectures and technologies related to *Isabelle* and *Isac* were explained and we saw how both projects combined *JVM*-based front-ends with logical engines developed in *Standard ML* and followed different approaches for the communication between these layers. The systems' designs show how one can benefit from the advantages and strengths of different programming paradigms and platforms within a single architecture. The results of the case study are promising and show that depending on the problem at hand, the achievable speedup can be close to a factor of the number of available processing cores, potentially enhanced even more by the use of simultaneous multithreading (hyper-threading).

5.1 Future Work

While *Isac*'s internal user session management has now been parallelized, multiple user GUIs communicate with the same standard input stream on the *Standard ML* side and this communication is unsynchronized. The mathematic engine writes to one single standard output stream. The write opera-

tion in use is provided and synchronized by *Isabelle*. However, since all GUIs read from the same stream, they need to filter out all messages not meant for them and thus waste resources, especially with growing numbers of simultaneous user sessions. The next important step is therefore a more flexible, efficient and robust communication model which would then allow the parallel session management to be stressed and also fine-tuned to perform well in practice. Other improvements to *Isac* include utilizing of *Isabelle*'s function package and code generator to simplify and speedup *Isac*'s programming language, adaptation of *Isabelle*'s capabilities with respect to floating point numbers and complex numbers, possibly adopt term nets for better performance of *Isac*'s rewrite engine. The latter plans for improvement of the mathematics-engine can be pursued independently from work on the front-end due to a stable interface in between. While *Isac*'s own front-end will be extended and improved, future development of the *Isabelle/jEdit* IDE may enable *Isac* to adopt it as its front-end in the long run.

The whole case study involved deeply invasive refactoring processes. The presence of automatic tools for this kind of tasks in *Standard ML* would have been desirable. While there has been related work on *Haskell* [13, 23, 55, 88] and *Erlang* [15, 55], there are, to my knowledge, no such projects specifically for *Standard ML* available. These could also help remove unused code and support convenient restructuring of modules.

Appendix A

Code Samples and Comments

A.1 Fibonacci Implementations

The Fibonacci numbers are an integer sequence given by

$$F_n = F_{n-1} + F_{n-2} \tag{A.1}$$

and

$$F_0 = 0, F_1 = 1\ . \tag{A.2}$$

Because this is a very simple, recursive definition, it was chosen for the demonstration of certain concepts in this thesis. However, it is very important to note that the straight forward implementation

```
fun fib 0 = 0
  | fib 1 = 1
  | fib x = fib (x - 1) + fib (x - 2),
```

which we presented as an example for pattern matching in *Standard ML* (page 7, section 2.1.2), is very inefficient because it has a runtime behavior of $O(F_n)$. An efficient version of the function,

```
fun fib' 0 = (0, 1)
  | fib' n = let
      val (x1, x2) = fib' (n - 1)
    in
      (x2, x1 + x2)
    end;
(*gets 2nd element of touple*)
fun fib n = fib' (n - 1) |> snd ,
```

shows linear runtime behavior. Annotations (section 3.2.2) were demonstrated using the Fibonacci function (page 19):

```
fun fib 0 = 0
  | fib 1 = 1
  | fib x = fib (x - 1) + par (fib (x - 2))
```

In practice, the gain from parallelism in this example would not just be marginal, the overhead for managing parallel execution would most likely result in a performance worse than the sequential version. Note that annotations of this kind can only work in a programming language following a lazy evaluation strategy because in an eager language like *Standard ML* the annotated expression would be evaluated before being passed to the hypothetical `par` evaluation. Futures have been discussed in detail throughout this thesis (sections 3.2.3, 3.4.2, 3.4.3, 4.1.1). The example on page 19 redefined `fib` with a future:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib x = let
        fun fib' () = fib (x - 2)
        val fibf = Future.fork fib'
      in fib (x - 1) + (Future.join fibf) end
```

This code does work in *Isabelle/ML* (section 4.1.1). The performance concerns we saw with the previous example also apply here: in practice, parallelizing the evaluation of the Fibonacci function in this way makes no sense.

A.2 `merge_lists` implementation

In the `Theory_Data` implementation on page 50 (section 4.3.1) we omitted the definitions of the functions `merge_lists` and `merge_lists'` for simplicity reasons. They have been added here (program A.1). Please note that they require that their input lists of type `int list` are already ordered. The difference is that `merge_lists` accepts the two lists in a touple and `merge_lists'` as two separate arguments.

A.3 Irrationality of $\sqrt{2}$ in *Isar*

Program A.2 is a simple proof in *Isar*, showing that the square root of 2 is an irrational number[1].

[1] `~~/src/HOL/ex/Sqrt.thy`

Program A.1: `merge_lists` implementation.

```
fun merge out [] xs = (rev out) @ xs
  | merge out xs [] = (rev out) @ xs
  | merge out (xs' as x::xs) (ys' as y::ys) =
      if x = y then
        merge (x::out) xs ys
      else if x < y then
        merge (x::out) xs ys'
      else
        merge (y::out) xs' ys;
val merge_lists' = merge [];
fun merge_lists (xs, ys) = merge_lists' xs ys;
```

Program A.2: *Isabelle/Isar* proof of the irrationality of $\sqrt{2}$.

```
lemma "∃a b::real. a ∉ ℚ ∧ b ∉ ℚ ∧ a powr b ∈ ℚ"
  (is "EX a b. ?P a b")
proof cases
  assume "sqrt 2 powr sqrt 2 ∈ ℚ"
  then have "?P (sqrt 2) (sqrt 2)"
    by (metis sqrt_2_not_rat)
  then show ?thesis by blast
next
  assume 1: "sqrt 2 powr sqrt 2 ∉ ℚ"
  have "(sqrt 2 powr sqrt 2) powr sqrt 2 = 2"
    using powr_realpow [of _ 2]
    by (simp add: powr_powr power2_eq_square [symmetric])
  then have "?P (sqrt 2 powr sqrt 2) (sqrt 2)"
    by (metis 1 Rats_number_of sqrt_2_not_rat)
  then show ?thesis by blast
qed
```

Appendix B

Content of the DVD

Format: DVD-R, Single Layer, ISO9660-Format

B.1 Master Thesis

Pfad: /thesis/

Lehnfeld14.pdf	Master thesis
images/*.png	Images
literature/*.pdf	Copies of online/selected sources

B.2 Project Files

Pfad: /project/

demos/	Demo theories, performance test
repo/	Repository snapshot (28/06/2014)
README.md	Usage instructions

References

Literature

[1] Peter Achten and Marinus J. Plasmeijer. “The Ins and Outs of Clean I/O”. In: *Journal of Functional Programming* 5.1 (1995), pp. 81–110 (cit. on p. 7).

[2] Gul A. Agha. “ACTORS - a model of concurrent computation in distributed systems”. In: MIT Press series in artificial intelligence (1990), pp. I–IX, 1–144 (cit. on p. 23).

[3] Gul A. Agha et al. “A Foundation for Actor Computation”. In: *Journal of Functional Programming* 7.1 (1997), pp. 1–72 (cit. on p. 23).

[4] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. (Atlantic City, New Jersey). AFIPS ’67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. URL: http://doi.acm.org/10.1145/1465482.1465560 (cit. on p. 11).

[5] Kenneth Appel and Wolfgang Haken. “The solution of the four-color-map problem”. In: *Scientific American* 237 (1977), pp. 108–121 (cit. on p. 9).

[6] David Aspinall. “Proof General: A Generic Tool for Proof Development”. In: *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*. TACAS ’00. London, UK, UK: Springer-Verlag, 2000, pp. 38–42. URL: http://dl.acm.org/citation.cfm?id=646484.691773 (cit. on p. 47).

[7] Ralph-Johan Back. “Structured derivations: a unified proof style for teaching mathematics”. In: *Formal Aspects of Computing* 22.5 (2010), pp. 629–661 (cit. on p. 41).

[8] John W. Backus. “Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs”. In: *Communications of the ACM* 21.8 (1978), pp. 613–641 (cit. on p. 5).

[9] Henry C. Baker Jr. and Carl Hewitt. “The Incremental Garbage Collection of Processes”. In: *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. URL: http://doi.acm.org/10.1145/800228.806932 (cit. on p. 19).

[10] L. S. van Benthem Jutting. *Checking Landau’s “Grundlagen” in the Automath system*. Eindhoven: Mathematisch Centrum, 1979 (cit. on p. 8).

[11] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. “Automatic Proof and Disproof in Isabelle/HOL”. In: *Proceedings of the 8th International Conference on Frontiers of Combining Systems*. (Saarbrücken, Germany). FroCoS’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 12–27. URL: http://dl.acm.org/citation.cfm?id=2050784.2050787 (cit. on p. 36).

[12] Guy E. Blelloch and Gary Sabot. “Compiling Collection-Oriented Languages onto Massively Parallel Computers”. In: *Journal of Parallel and Distributed Computing* 8.2 (1990), pp. 119–134 (cit. on p. 21).

[13] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. “ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques”. In: *Proceedings of the 12th International Conference on Trends in Functional Programming*. (Madrid, Spain). TFP’11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 82–97. URL: http://dx.doi.org/10.1007/978-3-642-32037-8_6 (cit. on pp. 25, 26, 56).

[14] Christopher Brown et al. “A Language-Independent Parallel Refactoring Framework”. In: *Proceedings of the Fifth Workshop on Refactoring Tools*. (Rapperswil, Switzerland). WRT ’12. New York, NY, USA: ACM, 2012, pp. 54–58. URL: http://doi.acm.org/10.1145/2328876.2328884 (cit. on p. 25).

[15] Christopher Brown et al. “Cost-Directed Refactoring for Parallel Erlang Programs”. In: *International Journal of Parallel Programming* 42.4 (2014), pp. 564–582 (cit. on pp. 25, 26, 56).

[16] Bruno Buchberger. *Mathematik für Informatiker II (Problemlösestrategien und Algorithmentypen)*. Lecture notes CAMP-Publ.-No. 84-4.0. RISC-Linz, 1984 (cit. on p. 45).

[17] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975 (cit. on p. 1).

[18] Eugene Charniak et al. *Artificial Intelligence Programming*. 2nd ed. Psychology Press, 1987 (cit. on p. 47).

[19] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013 (cit. on pp. 9, 34).

[20] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of mathematics* 33.2 (1932), pp. 346–366 (cit. on p. 5).

[21] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture - a hardware / software approach*. Morgan Kaufmann, 1999, pp. I–XXIX, 1–1024 (cit. on p. 14).

[22] Gabriella Daróczy and Walther Neuper. "Error-Patterns within "Next-Step-Guidance" in TP-based Educational Systems". In: *Electronic Journal of Mathematics & Technology* 7.2 (2013) (cit. on pp. 43, 44).

[23] Danny Dig. "A Refactoring Approach to Parallelism". In: *IEEE Software* 28.1 (2011), pp. 17–22 (cit. on pp. 25, 56).

[24] Sean Dorward, Rob Pike, and Phil Winterbottom. "Programming in Limbo". In: *Proceedings of the 42Nd IEEE International Computer Conference*. COMPCON '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 245–. URL: http://dl.acm.org/citation.cfm?id=792770.793719 (cit. on p. 25).

[25] Robert Ennals. *Software transactional memory should not be obstruction-free*. Tech. rep. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, 2006 (cit. on p. 22).

[26] John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. "Vienna Development Method". In: *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2007. URL: http://dx.doi.org/10.1002/9780470050118.ecse447 (cit. on p. 10).

[27] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. "Understanding Code Mobility". In: *IEEE Transactions on Software Engineering* 24.5 (1998), pp. 342–361 (cit. on p. 23).

[28] David Geer. "Industry Trends: Chip Makers Turn to Multicore Processors". In: *IEEE Computer* 38.5 (2005), pp. 11–13 (cit. on pp. 1, 13).

[29] Georges Gonthier. "Formal Proof — The Four Colour Theorem". In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393 (cit. on p. 9).

[30] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher-order logic*. Cambridge University Press, 1993 (cit. on p. 9).

[31] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Communications of the ACM* 31.5 (1988), pp. 532–533 (cit. on p. 11).

[32] Florian Haftmann and Tobias Nipkow. “Code Generation via Higher-order Rewrite Systems”. In: *Proceedings of the 10th International Conference on Functional and Logic Programming*. (Sendai, Japan). FLOPS’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 103–117. URL: http://dx.doi.org/10.1007/978-3-642-12251-4_9 (cit. on pp. 11, 47).

[33] Thomas C. Hales. “Historical Overview of the Kepler Conjecture”. English. In: *The Kepler Conjecture*. Ed. by Jeffrey C. Lagarias. Springer New York, 2011, pp. 65–82. URL: http://dx.doi.org/10.1007/978-1-4614-1129-1_3 (cit. on p. 10).

[34] Philipp Haller. “An Object-Oriented Programming Model for Event-Based Actors”. MA thesis. University of Karlsruhe, 2006 (cit. on p. 39).

[35] Matthew Hammer et al. “A Proposal for Parallel Self-adjusting Computation”. In: *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*. (Nice, France). DAMP ’07. New York, NY, USA: ACM, 2007, pp. 3–9. URL: http://doi.acm.org/10.1145/1248648.1248651 (cit. on p. 18).

[36] Kevin Hammond and Greg Michelson, eds. *Research Directions in Parallel Functional Programming*. London, UK, UK: Springer-Verlag, 2000 (cit. on pp. 4, 21).

[37] Kevin Hammond et al. “The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems”. English. In: *Formal Methods for Components and Objects*. Ed. by Bernhard Beckert et al. Vol. 7542. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 218–236. URL: http://dx.doi.org/10.1007/978-3-642-35887-6_12 (cit. on p. 25).

[38] Tim Harris and Satnam Singh. “Feedback Directed Implicit Parallelism”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. (Freiburg, Germany). ICFP ’07. New York, NY, USA: ACM, 2007, pp. 251–264. URL: http://doi.acm.org/10.1145/1291151.1291192 (cit. on p. 14).

[39] Tim Harris et al. “Composable Memory Transactions”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. (Chicago, IL, USA). PPoPP ’05. New York, NY, USA: ACM, 2005, pp. 48–60. URL: http://doi.acm.org/10.1145/1065944.1065952 (cit. on p. 22).

[40] Peter B. Henderson. “Functional Programming, Formal Specification, and Rapid Prototyping”. In: *IEEE Transactions on Software Engineering* 12.2 (1986), pp. 241–250 (cit. on p. 14).

[41] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)* Morgan Kaufmann, 2012 (cit. on p. 14).

[42] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. URL: http://doi.acm.org/10.1145/78969.78972 (cit. on p. 22).

[43] Carl Hewitt, Peter Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. (Stanford, USA). IJCAI’73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804 (cit. on p. 22).

[44] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Communications of the ACM* 21.8 (1978), pp. 666–677 (cit. on p. 24).

[45] Paul Hudak. “Conception, Evolution, and Application of Functional Programming Languages”. In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 359–411. URL: http://doi.acm.org/10.1145/72551.72554 (cit. on pp. 4, 6).

[46] INMOS, Ltd. *The occam programming manual*. Englewood Cliffs: Prentice Hall, 1984 (cit. on p. 25).

[47] Robert H. Halstead Jr. “Multilisp: A Language for Concurrent Symbolic Computation”. In: *ACM Transactions on Programming Languages and Systems* 7.4 (1985), pp. 501–538 (cit. on p. 19).

[48] Rajesh K. Karmani, Amin Shali, and Gul A. Agha. “Actor Frameworks for the JVM Platform: A Comparative Analysis”. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. (Calgary, Alberta, Canada). PPPJ ’09. New York, NY, USA: ACM, 2009, pp. 11–20. URL: http://doi.acm.org/10.1145/1596655.1596658 (cit. on p. 23).

[49] Ken Kennedy, Kathryn S. McKinley, and Chau-Wen Tseng. “Interactive Parallel Programming using the ParaScope Editor”. In: *IEEE Transactions on Parallel and Distributed Systems* 2.3 (1991), pp. 329–341 (cit. on p. 25).

[50] Markus Kienleitner. “Towards “NextStep Userguidance” in a Mechanized Math Assistant”. Bachelor’s thesis. Graz University of Technology, 2012 (cit. on p. 43).

[51] Alexander Krauss. *Defining recursive functions in Isabelle/HOL*. 2013. URL: http://isabelle.in.tum.de/doc/functions.pdf (cit. on p. 47).

[52] Peter J. Landin. "The next 700 programming languages". In: *Communications of the ACM* 9.3 (1966), pp. 157–166 (cit. on p. 9).

[53] John Launchbury. "A Natural Semantics for Lazy Evaluation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. (Charleston, South Carolina, USA). POPL '93. New York, NY, USA: ACM, 1993, pp. 144–154. URL: http://doi.acm.org/10.1145/158511.158618 (cit. on p. 7).

[54] Mathias Lehnfeld. "Verbindung von 'Computation' und 'Deduction' im *Isac*-System". Bachelor's project report. Technische Universität Wien, 2011 (cit. on p. 46).

[55] Huiqing Li and Simon Thompson. "Comparative Study of Refactoring Haskell and Erlang Programs". In: *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. SCAM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 197–206. URL: http://dx.doi.org/10.1109/SCAM.2006.8 (cit. on pp. 26, 56).

[56] Ben Lippmeier. "Type Inference and Optimisation for an Impure World". PhD thesis. Australian National University, 2009 (cit. on p. 7).

[57] Simon Marlow, Simon Peyton Jones, and Satnam Singh. "Runtime Support for Multicore Haskell". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. (Edinburgh, Scotland). ICFP '09. New York, NY, USA: ACM, 2009, pp. 65–78. URL: http://doi.acm.org/10.1145/1596550.1596563 (cit. on p. 20).

[58] Daniel Matichuk, Makarius Wenzel, and Toby Murray. "An Isabelle Proof Method Language". In: *Proceedings of the 5th Int. Conference on Interactive Theorem Proving*. (Vienna, Austria). Ed. by G Klein and R Gamboa. Springer-Verlag, July 2014 (cit. on p. 33).

[59] David C. J. Matthews. *Papers on Poly/ML*. Tech. rep. Computer Laboratory, University of Cambridge, Feb. 1989 (cit. on p. 29).

[60] David C. J. Matthews and Makarius Wenzel. "Efficient Parallel Programming in Poly/ML and Isabelle/ML". In: *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. (Madrid, Spain). DAMP '10. New York, NY, USA: ACM, 2010, pp. 53–62. URL: http://doi.acm.org/10.1145/1708046.1708058 (cit. on pp. 19, 30–32, 39).

[61] John McCarthy. “History of LISP”. In: *SIGPLAN Not.* 13.8 (Aug. 1978), pp. 217–223. URL: http://doi.acm.org/10.1145/960118.808387 (cit. on p. 5).

[62] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375 (cit. on pp. 5, 8, 29).

[63] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997 (cit. on pp. 5, 16, 28–30).

[64] Cleve Moler. “Matrix computation on distributed memory multiprocessors”. In: *Hypercube Multiprocessors* 86 (1986), pp. 181–195 (cit. on p. 12).

[65] Gordon E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (Apr. 19, 1965), pp. 114–117 (cit. on pp. 1, 12).

[66] Samuel K. Moore. “Multicore is bad news for supercomputers”. In: *Spectrum, IEEE* 45.11 (Nov. 2008), p. 15 (cit. on p. 13).

[67] Walther Neuper. “Automated Generation of User Guidance by Combining Computation and Deduction”. In: *Proceedings of the First Workshop on CTP Components for Educational Software (THedu’11)*. (Wroclaw, Poland). Ed. by Pedro Quaresma and Ralph-Johan Back. Vol. 79. EPTCS. July 2011, pp. 82–101 (cit. on pp. 2, 42, 46).

[68] John Nickolls and William J. Dally. “The GPU Computing Era”. In: *IEEE Micro* 30.2 (2010), pp. 56–69 (cit. on pp. 1, 21).

[69] Tobias Nipkow, Makarius Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Dec. 2013 (cit. on pp. 1, 9).

[70] Eric Nöcker et al. “Concurrent Clean”. In: *Proceedings on Parallel Architectures and Languages Europe: Volume II: Parallel Languages*. (Eindhoven, The Netherlands). PARLE ’91. New York, NY, USA: Springer-Verlag New York, Inc., 1991, pp. 202–219. URL: http://dl.acm.org/citation.cfm?id=111634.111646 (cit. on p. 19).

[71] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. LAMP-REPORT-2006-001. 2nd edition. 1015 Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne, 2006 (cit. on p. 24).

[72] Rajendra Panwar and Gul A. Agha. “A Methodology for Programming Scalable Architectures”. In: *Journal of Parallel and Distributed Computing* 22.3 (1994), pp. 479–487 (cit. on p. 23).

[73] Lawrence C. Paulson. "Isabelle: The Next Seven Hundred Theorem Provers". In: *Proceedings of the 9th International Conference on Automated Deduction*. London, UK, UK: Springer-Verlag, 1988, pp. 772–773. URL: http://dl.acm.org/citation.cfm?id=648228.751990 (cit. on p. 9).

[74] Simon Peyton Jones. "Harnessing the Multicores: Nested Data Parallelism in Haskell". In: *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. (Bangalore, India). APLAS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 138–138. URL: http://dx.doi.org/10.1007/978-3-540-89330-1_10 (cit. on p. 21).

[75] Simon Peyton Jones et al. "The Haskell 98 Language and Libraries: The Revised Report". In: *Journal of Functional Programming* 13.1 (Jan. 2003), pp. –255. URL: http://www.haskell.org/definition/ (cit. on pp. 5, 7).

[76] Rob Pike. "Go at Google". In: *Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity*. (Tucson, AZ, USA). Ed. by Gary T. Leavens. 2012, pp. 5–6 (cit. on p. 25).

[77] Hany E. Ramadan et al. "Committing Conflicting Transactions in an STM". In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. (Raleigh, NC, USA). PPoPP '09. New York, NY, USA: ACM, 2009, pp. 163–172. URL: http://doi.acm.org/10.1145/1504176.1504201 (cit. on p. 22).

[78] John H. Reppy. "Concurrent ML: Design, Application and Semantics". In: *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992*. (McMaster University, Hamilton, Ontario, Canada). Springer-Verlag, 1993, pp. 165–198 (cit. on p. 30).

[79] Jan Rocnik. "Interactive Course Material for Signal Processing based on Isabelle/*Isac*". Bachelor's thesis. Graz University of Technology, 2012. URL: http://www.ist.tugraz.at/projects/isac/publ/jrocnik_bakk.pdf (cit. on p. 47).

[80] Bratin Saha et al. "McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime". In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '06. ACM, 2006, pp. 187–197 (cit. on p. 22).

[81] H. P. Sharangpani and M. L. Barton. *Statistical Analysis of Floating Point Flaw in the Pentium Processor*. Tech. rep. Intel Corporation, 1994 (cit. on p. 10).

[82] Nir Shavit and Dan Touitou. "Software transactional memory". In: *Distributed Computing* 10.2 (1997), pp. 99–116 (cit. on p. 22).

[83] Herbert A. Simon. *The Shape of Automation for Men and Management*. New York: Harper & Row, 1965 (cit. on p. 8).

[84] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's Journal* 30.3 (2005), pp. 202–210. URL: http://www.gotw.ca/publications/concurrency-ddj.htm (cit. on pp. 1, 2).

[85] Herb Sutter and James R. Larus. "Software and the concurrency revolution". In: *ACM Queue* 3.7 (2005), pp. 54–62 (cit. on pp. 1, 15).

[86] David Tarditi, Sidd Puri, and Jose Oglesby. "Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses". In: *SIGOPS Oper. Syst. Rev.* 40.5 (Oct. 2006), pp. 325–335 (cit. on p. 21).

[87] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. (Montpellier, France). ECOOP'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 302–326. URL: http://dx.doi.org/10.1007/978-3-642-39038-8_13 (cit. on p. 24).

[88] Simon Thompson. "Refactoring Functional Programs". In: *Proceedings of the 5th International Conference on Advanced Functional Programming*. (Tartu, Estonia). AFP'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 331–357. URL: http://dx.doi.org/10.1007/11546382_9 (cit. on pp. 15, 16, 56).

[89] Martin Thurau. *Akka framework*. Paper for Seminar Software Systems Engineering, Universität zu Lübeck. 2012. URL: http://media.itm.uni-luebeck.de/teaching/ws2012/sem-sse/martin-thurau-akka.io.pdf (cit. on p. 23).

[90] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. "Simultaneous Multithreading: Maximizing On-chip Parallelism". In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. (S. Margherita Ligure, Italy). ISCA '95. New York, NY, USA: ACM, 1995, pp. 392–403. URL: http://doi.acm.org/10.1145/223982.224449 (cit. on p. 14).

[91] Stephen Weeks. "Whole-program Compilation in MLton". In: *Proceedings of the 2006 Workshop on ML*. (Portland, Oregon, USA). ML '06. New York, NY, USA: ACM, 2006, pp. 1–1. URL: http://doi.acm.org/10.1145/1159876.1159877 (cit. on p. 29).

[92] Makarius Wenzel. "Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit". In: *Electronic Notes in Theoretical Computer Science* 285 (2012), pp. 101–114 (cit. on pp. 34, 39, 45).

[93] Makarius Wenzel. "Asynchronous User Interaction and Tool Integration in Isabelle/PIDE". In: *Proceedings of the 5th Int. Conference on Interactive Theorem Proving*. (Vienna, Austria). Ed. by G Klein and R Gamboa. July 2014 (cit. on p. 35).

[94] Makarius Wenzel. "Isabelle As Document-Oriented Proof Assistant". In: *Proceedings of the 18th Calculemus and 10th International Conference on Intelligent Computer Mathematics*. (Bertinoro, Italy). MKM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 244–259. URL: http://dl.acm.org/citation.cfm?id=2032713.2032732 (cit. on pp. 35, 39).

[95] Makarius Wenzel. "Isabelle/jEdit: A Prover IDE Within the PIDE Framework". In: *Proceedings of the 11th International Conference on Intelligent Computer Mathematics*. (Bremen, Germany). CICM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 468–471. URL: http://dx.doi.org/10.1007/978-3-642-31374-5_38 (cit. on pp. 35, 39).

[96] Makarius Wenzel. "Isar — A Generic Interpretative Approach to Readable Formal Proof Documents". In: *Theorem Proving in Higher Order Logics*. Ed. by Yves Bertot et al. Vol. 1690. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 167–183 (cit. on pp. 32, 33, 39).

[97] Makarius Wenzel. "Parallel Proof Checking in Isabelle/Isar". In: *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*. Ed. by G Dos Reis and L Théry. ACM Digital Library, 2009 (cit. on pp. 36, 45).

[98] Makarius Wenzel. *The Isabelle/Isar Implementation*. With contributions by Florian Haftmann and Larry Paulson. 2013. URL: http://isabelle.in.tum.de/doc/implementation.pdf (cit. on pp. 30, 39, 46, 49).

[99] Makarius Wenzel et al. *The Isabelle/Isar Reference Manual*. 2013. URL: http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/isar-ref.pdf (cit. on pp. 32, 50).

[100] Jan Wloka, Manu Sridharan, and Frank Tip. "Refactoring for Reentrancy". In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. (Amsterdam, The Netherlands). ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 173–182. URL: http://doi.acm.org/10.1145/1595696.1595723 (cit. on p. 25).

[101] Lingli Zhang et al. "Software Transaction Commit Order and Conflict Management". US 7711678. May 2010. URL: http://www.google.co.in/patents/US7711678 (cit. on p. 22).

Online sources

[102] *Documentation*. 2013. URL: http://isabelle.in.tum.de/documentation.html (visited on 06/13/2014) (cit. on pp. 9, 37).

[103] *E-Math*. 2013. URL: http://emath.eu/en/ (visited on 06/05/2014) (cit. on p. 41).

[104] *FlyspeckFactSheet*. 2012. URL: https://code.google.com/p/flyspeck/wiki/FlyspeckFactSheet (visited on 06/01/2014) (cit. on p. 10).

[105] *Functional programming - HaskellWiki*. 2013. URL: http://www.haskell.org/haskellwiki/Functional_programming (cit. on p. 6).

[106] *Futures (Scala) – Akka Documentation*. 2014. URL: http://doc.akka.io/docs/akka/current/scala/futures.html (visited on 06/05/2014) (cit. on pp. 20, 24).

[107] John Harrison. *HOL Light*. URL: http://www.cl.cam.ac.uk/~jrh13/hol-light/ (visited on 05/26/2014) (cit. on p. 9).

[108] *Installation*. 2013. URL: http://isabelle.in.tum.de/installation.html (visited on 06/05/2014) (cit. on p. 28).

[109] *Intel® Pentium® Processor - Invalid Instruction Erratum Overview*. 1997. URL: http://www.intel.com/support/processors/pentium/ppiie/index.htm (visited on 05/26/2014) (cit. on p. 10).

[110] *isabelle: Summary*. 2014. URL: http://isabelle.in.tum.de/repos/isabelle/ (visited on 06/21/2014) (cit. on p. 38).

[111] *Isabelle/Eclipse*. 2013. URL: http://andriusvelykis.github.io/isabelle-eclipse/ (visited on 06/05/2014) (cit. on p. 40).

[112] *Isabelle/HOL sessions*. URL: http://isabelle.in.tum.de/dist/library/HOL/ (visited on 06/13/2014) (cit. on p. 42).

[113] *isabelle.in.tum.de/dist/Isabelle2013-2/NEWS*. 2013. URL: http://isabelle.in.tum.de/dist/Isabelle2013-2/NEWS (visited on 05/27/2014) (cit. on pp. 46, 48).

[114] Vojin Jovanovic and Philipp Haller. *The Scala Actors Migration Guide - Scala Documentation*. 2014. URL: http://docs.scala-lang.org/overviews/core/actors-migration-guide.html (cit. on p. 24).

[115] *Moscow ML*. 2014-05-15. URL: http://mosml.org/ (cit. on p. 29).

[116] The SML/NJ Fellowship. *Standard ML of New Jersey*. URL: http://www.smlnj.org/ (visited on 05/15/2014) (cit. on p. 29).

[117] *The Cl-isabelle-users Archives*. 2014. URL: https://lists.cam.ac.uk/pipermail/cl-isabelle-users/ (visited on 06/05/2014) (cit. on p. 28).

[118] *The Team*. 2013. URL: http://www21.in.tum.de/team (visited on 06/05/2014) (cit. on p. 28).

[119] Makarius Wenzel. *Re: [isabelle-dev] scala-2.11.0*. 2014. URL: http://www.mail-archive.com/isabelle-dev@mailbroy.informatik.tu-muenchen.de/msg05253.html (visited on 06/16/2014) (cit. on p. 34).

[120] *www.ist.tugraz.at/projects/isac/www/kbase/met/index_met.html*. URL: http://www.ist.tugraz.at/projects/isac/www/kbase/met/index_met.html (visited on 06/01/2014) (cit. on p. 45).

[121] *www.ist.tugraz.at/projects/isac/www/kbase/pbl/index_pbl.html*. URL: http://www.ist.tugraz.at/projects/isac/www/kbase/pbl/index_pbl.html (visited on 06/01/2014) (cit. on p. 45).

[122] *www.ist.tugraz.at/projects/isac/www/kbase/pbl/pbl_equ_univ.html*. URL: http://www.ist.tugraz.at/projects/isac/www/kbase/pbl/pbl_equ_univ.html (visited on 06/13/2014) (cit. on p. 45).

[123] *www.ist.tugraz.at/projects/isac/www/kbase/thy/index_thy.html*. URL: http://www.ist.tugraz.at/projects/isac/www/kbase/thy/index_thy.html (visited on 06/01/2014) (cit. on p. 45).

[124] *Zemanek, Heinz | Austria-Forum > Biographien*. URL: http://austria-forum.org/af/Wissenssammlungen/Biographien/Zemanek,_Heinz (visited on 06/02/2014) (cit. on p. 10).