JOHANNES KEPLER

UNIVERSITÄT LINZ

Netzwerk für Forschung, Lehre und Praxis

Compilation of Theorema Programs

DISSERTATION

zur Erlangung des akademischen Grades

DOKTOR DER TECHNISCHEN WISSENSCHAFTEN

Angefertigt am *Institut für Symbolisches Rechnen*

Betreuung:

Erster Begutachter: *o.Univ.-Prof. Dr. Dr.h.c.mult. Bruno Buchberger*

Zweiter Begutachter: *a.Univ.-Prof. Dr. Josef Küng*

Eingereicht von:

Dipl.-Ing. Alexander Zapletal

Linz, Mai 2008

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Mai 2008

Alexander Zapletal

Abstract

In this thesis we present a compiler which is able to translate Theorema programs into executable Java code, which can then be used for extensive and fast calculations called from within Theorema.

Generally, it can be observed that higher elegance in programming languages and software systems must be paid for by dramatically increasing computing times, see for example Prolog computations and original Theorema. One of the basic strategical goals of the Theorema system is to offer predicate logic as a uniform frame for the three main activities of mathematics: proving, solving, and computing. It is one of the strong features of Theorema that it combines automated theorem proving *and* computation in one logical and software frame. In fact, the same Theorema definitions that are used for stating and proving theorems can also be applied for computing.

The actual motivation for this thesis was the slowness of computations in the current version of Theorema, which is due to the usage of special logical inference rules (directed equational logic) as an interpreter for the Theorema algorithms. Therefore, it is of utmost importance to find a way to drastically speed-up the execution of Theorema algorithms without losing the elegance of writing the algorithms in the same predicate logic version (namely that of Theorema) in which also general mathematical statements, in particular correctness theorems for algorithms, are expressed. The main approach for achieving this goal is compilation of Theorema algorithms into a machine-oriented language, in our case Java. It turns out that this is possible for Theorema algorithms, at least for a well defined and rich class of practically interesting algorithms that includes the full power of induction, sequence variables, and even functors.

In this thesis we will show how this goal of compilation of Theorema programs can be achieved in a satisfactory way that brings the execution times of compiled Theorema programs drastically below the execution times of Mathematica algorithms and not more than a factor of 100 above the execution times of hand coded Java algorithms.

Keywords: Compilation, Predicate Logic, Theorema

Zusammenfassung

In dieser Dissertation wird ein Compiler vorgestellt, der Theorema-Programme in ausführbaren Java Code übersetzen kann. Dieser Code kann dann für schnelle Berechnungen von Theorema aus exekutiert werden.

Höhere Eleganz bei Programmiersprachen und Softwaresystemen muss in der Praxis meist mit dramatisch schlechteren Laufzeiten teuer bezahlt werden, siehe Prolog und (die derzeitige Version von) Theorema. Eines der zentralen und grundlegenden Ziele von Theorema ist der Einsatz von Prädikatenlogik als ein einheitliches System für die drei Hauptaktivitäten in der Mathematik: Beweisen, Lösen und Berechnen. Eine der herausragenden Besonderheiten von Theorema ist die Kombination von automatischem Beweisen *und* Berechnungen in einem logischen und softwaretechnischen Rahmen. Tatsächlich können die Theorema-Definitionen, die zum Formulieren und Beweisen von Theoremen verwendet werden, auch für Berechnungen angewendet werden.

Die eigentliche Motivation für diese Arbeit war die Langsamkeit von Berechnungen in der derzeitigen Version von Theorema, die durch die Verwendung von speziellen logischen Schlussregeln (gerichtete Gleichheitslogik) als Interpreter für Theorema-Algorithmen bedingt ist. Daher ist es von größter Wichtigkeit einen Weg zu finden, die Ausführung von Theorema-Algorithmen drastisch zu beschleunigen, ohne jedoch die Eleganz zu verlieren, die Algorithmen in der selben Prädikatenlogikversion (nämlich jener von Theorema) zu schreiben, in der auch generelle mathematische Aussagen, insbesondere Korrektheitsbeweise von Algorithmen, formuliert sind. Der zentrale Ansatz zur Erreichung dieses Ziels ist die Kompilierung von Theorema-Algorithmen in eine maschinenorientierte Sprache, in unserem Fall Java. Es stellt sich heraus, dass das zumindest für eine wohldefinierte and reichhaltige Klasse von in der Praxis interessanten Algorithmen möglich ist, die vor allem auch Induktion, Sequenzvariablen und sogar Funktoren umfassen können.

In dieser Arbeit zeigen wir, wie das Ziel der Kompilierung von Theorema-Programmen auf eine zufriedenstellende Weise erreicht werden kann, sodass die Ausführungszeiten von kompilierten Theorema-Programmen deutlich unter jenen von Mathematica und nur um einen Faktor 100 über jenen von direkt in Java geschriebenen Algorithmen liegen.

Schlüsselwörter: Kompilation, Prädikatenlogik, Theorema

Acknowledgement

First of all, I want to thank Professor Bruno Buchberger, my advisor, for giving me the opportunity to study at RICAM and at RISC, in such a wonderful place like Hagenberg. Moreover, I want to thank him for his encouraging and always stimulating support in all of our Theorema seminars and special meetings. Throughout my whole Ph.D. time and the development of this work he was an inspiring guide who always had the right idea at the right moment. I also want to express my gratefulness to him, Professor Heinz W. Engl, and Professor Franz Winkler for making my financial support during my studies in Linz and Hagenberg possible.

Further, I want to thank all my colleagues from RICAM and RISC, especially Wolfgang Windsteiger and Laura Kovacs, for all their help and support.

Finally, I want to cordially thank my family for all their support and patience throughout my studies.

Contents

Introduction

In this thesis we present a compiler for the Theorema system. It is able to translate Theorema programs into executable Java byte code, which can then be used for extensive and fast calculations called from within Theorema. It is one of the strong features of the Theorema system that it combines automated theorem proving *and* computation in one logical and software frame. In fact, the same Theorema definitions that are used for stating and proving theorems can also be applied for computing.

The actual motivation for this thesis was the slowness of computations in the current version of Theorema, which is due to the usage of special logical inference rules (directed equational logic) as an interpreter for the Theorema algorithms. Especially when working with functors and combining them to nested "towers", computations become so slow that they are only interesting for pedagogical purposes but not for actual scientific applications of Theorema. Therefore, we wanted to come up with an approach to drastically speed-up computation times in Theorema, and the compilation to a fast and modern language like Java is the natural way to achieve this goal.

Two aspects were the driving principles during the design and development of the Theorema-Java Compiler presented in this thesis:

- All programs formulated in the current Theorema language should be translatable by the compiler. This includes predicate logic quantifiers with bounded range (e.g., $\forall$ and $\exists$), special Theorema quantifiers (e.g., the TupleOf quantifier and the SumOf quantifier), sequence variables, and, particularly, functors.

- Computing with the compiled Theorema programs should be completely hidden from the user, i.e., it should not be necessary for the user to get in contact with the Java code. Nevertheless, the user is, of course, able to access the well readable and well structured Java source code.

Combination of Elegance and Efficiency

Generally, it can be observed that higher elegance in programming languages and software systems must be paid for by dramatically increasing computing times, see for example Prolog computations and original Theorema. One of the basic strategical goals of the Theorema system is to offer predicate logic as a uniform frame for the three main activities of mathematics: proving, solving, and computing, see [Buch97], [Buch99c], [Buch00], [Buch04]. In particular, computing, in this view, is just a special case of proving, namely proving by conditional rewriting of ground terms. Thus, exploration sequences of the following kind should be possible in Theorema:

- Specify a problem, e.g., the computation of Gröbner Bases,

- Propose an algorithm for the solution of the problem, e.g., Buchberger's algorithm,

- Prove the correctness of the algorithm,

- Compute by applying the correct algorithm to concrete input.

This exploration sequence is possible in Theorema since its design and implementation in 1996 ([Buch96b], [Buch96c], [Buch96d], [Buch96e], [Buch97], [Tma97], [Tma98], [Tma00], [Tma06], [WiBu06]). Both the Theorema reasoners and the Theorema "computers" are written in the same meta language, namely Mathematica. Not surprisingly, considering computing as special proving leads to intolerably slow execution of algorithms. In fact, computations in Theorema can not be faster than computations in the Mathematica language, which by itself is slow, see [Buch91]. In practice, it is even slower by some constant, but not dramatic, factor. Thus, the current Theorema interpreter for Theorema algorithms has only pedagogical value. But sometimes even the pedagogical goals can not be achieved because running times are too long even for very small examples, and so, for example, the study of computing time behavior of various version of an algorithm can not be explored in the classroom context.

Therefore, it is of utmost importance to find a way to drastically speed-up the execution of Theorema algorithms without losing the elegance of writing the algorithm in the same Theorema predicate logic version in which also general mathematical statements, in particular correctness theorems for algorithms, are expressed.

The main approach for achieving this goal is the compilation of Theorema algorithms into a machine-oriented language, like C, C++, or Java (see Section 2.1). It turns out that this is possible for Theorema algorithms, at least for a well defined and rich class of practically interesting algorithms that includes the full power of induction, sequence variables, and even functors. Note that, in contrast, compilation is not possible in full-fledged Mathematica because of its many ad hoc peculiarities.

In this thesis we will show how this goal of compilation of Theorema programs can be achieved in a satisfactory way.

Statement of Originality

Theorema, developed at the Research Institute for Symbolic Computation (RISC, Johannes Kepler University Linz, Austria) since 1994, is implemented on the basis of the commercially distributed symbolic computation software system Mathematica and is based on a concept and on the ideas of Bruno Buchberger (see [Buch96b], [Buch96c], [Buch96d], [Buch96e], [Buch97]). Since the start of the Theorema project a lot of people, some of them left RISC, some are still active members in the Theorema group, have contributed to the constant development and improvement of the system. In the following list I will enumerate the most important members of the Theorema group as well as those who contributed most to the development of the compiler presented in the course of this thesis.

- Bruno Buchberger is the inventor of the Theorema system and also implemented its first version including the first prototypes of some provers. Especially, he introduced the concept of functors in Theorema (see [Tma00]), which is the very concept to build-up mathematics bottom-up, starting from simple domains (for example, rational numbers with simple operations like addition, multiplication) and repeatedly applying suitable functors to arrive at arbitrary complex domains. He is a very active member and the driving force of the Theorema group ever since the start of the Theorema project.

- Bruno Buchberger, again. Since he is not only the creator of the Theorema system, as stated above, but also the scientific supervisor of this thesis, it is, no doubt, appropriate mentioning him twice. He lively contributed to the content in many seminars and personal meetings. Particularly, the translation of functors (see Chapter 7) and of sequence variables (see Chapter 5) are based on his ideas ([Buch07a]).

- Tudor Jebelean, co-leader of the Theorema group, contributed to a module for the elimination of pattern matching (see Section 3.1), which is needed in the translation of Theorema theories into an intermediate code, which was designed by him as well ([Jebe07]).

- Martin Giese gave many very important advice and contributions in the starting phase of the development of the compiler. Particularly, the compilation of abstract data types and the associated class design (see Chapter 4) are mainly based on his ideas ([Gies07]).

- Wolfgang Windsteiger is the member of the Theorema group who has the best overview of the current implementation of the entire Theorema system. Therefore, he is in charge of the maintenance of the system and is also very active in the its further development. Moreover, he is a very helpful person always taking time to explain the internals of Theorema to newcomers of the group and helping to solve problems with it, however hard and involved they are. Without his cordial and extensive help this thesis would have been hardly possible.

- Temur Kutsia, also a very active member of the Theorema group, contributed to the translation of sequence variables by virtue of his rich experience in this area ([Kuts07]).

Structure of the Thesis

This thesis is divided into two main parts: a detailed description of the Theorema-Java Compiler and case studies showing the compiler in action. In the first part, we will give an exact description of the concrete implementation problems, which naturally arise from the inherent differences between the Theorema language and the Java language. Then, we will state the reasons why we chose Java as the target language of the compiler and also give a list of some special features of the compiler. The next section provides a first example, namely merging two sorted lists, and concretely shows how a Theorema theory can be compiled into Java code and how the thereby created code can be executed from within Theorema. Chapters 4-7 deal with the details of the translation of Theorema programs into Java byte code. Chapter 8 shows how to call compiled algorithms, and Chapter 9 documents commands the change the compiler's behavior. The final Chapter 10 describes the whole Java-sided framework, which provides several auxiliary Java classes.

Part 2 contains two case studies which demonstrate the capabilities and the usage of the Theorema-Java Compiler. The first case study (see Chapter 11), which is based on the work of Bruno Buchberger in [Buch03], is on computations of Gröbner Bases and shows the power of functors and their practical application. The second case study (see Chapter 12) presents a Theorema implementation of an algorithm for interpolating univariate polynomials and is based on [Wind06].

On the Document

This thesis has been created using the Mathematica 6.0 front end, and it exists in two version: an electronic version and a printed version. While the latter one is the classical form of a Ph.D. thesis, the electronic versions comes up with two main advantages:

- It offers the actual evaluation of input. This makes it possible to actually try out the presented examples and also to modify them and make one's own experiments.

- It uses hyperlinks to quickly jump from one part of the thesis to another.

Please note that, one needs Mathematica or at least MathReader for accessing the electronic version.

Apart from floating text, two main formatting styles of cells can be found in this thesis: input cells can be evaluated in the electronic version and look like this:

```
18 + 7
```

```
25
```

Cells which present Java code appear as a grey box, for instance:

```
int a = 18;
int b = 7;
System.out.println(a+b);
```

Part 1

The Theorema-Java Compiler

1 Computations in the Current Theorema System

In this part of the thesis we want to give a general overview of the Theorema system with a focus on its computational capabilities. After a short description of the whole system, we will concentrate on the way computing can be done in the current system. We will show the use of the two classical ways of computations in the Theorema system, namely the simple **`Compute`** command and the more advanced **`ComputationalSession`** command. The core issue of this thesis is the presentation of a new way to compute in Theorema by translating one's definitions into executable Java code and calling these compiled and optimized algorithms from within Theorema. For this, the framework of the Theorema-Java Compiler provides several new commands (especially **`Java–Compute`**), which will be presented in Chapter 8.

1.1 The Theorema System

The main philosophy of the Theorema system is to provide one logical and software system frame for the entire mathematical exploration process, that includes the formulation of concepts, the mathematical study of their properties, the formulation of mathematical problems, their solution by algorithms, the application of algorithms to concrete data ("computation"), and the systematic documentation of the exploration results in well structured knowledge bases, see [Buch96b], [Buch96c], [Buch96d], [Buch96e], [Buch97], [Buch99c], [Buch00], [Buch04]. In particular, the user of a system like Theorema need not switch between two systems when changing from proving to programming, or from searching in knowledge bases to checking the correctness of mathematical statements.

Theorema is built on top of Mathematica ([Mma]), a popular computer algebra system developed by Stephen Wolfram. More specifically, for keeping Theorema logically self-contained, only the programming language of Mathematica is used for the implementation of Theorema, no usage is made of Mathematica's algorithm library (except if the Theorema user explicitly access algorithms from this library). Theorema is currently an add-on package to Mathematica and can be loaded with the following command:

```
Needs["Theorema`"]
```

Mathematica, and hence Theorema as well, is currently supported by a wide range of computer systems: Windows, Linux, and Mac OS. It also provides an interface to Java, the so-called J/Link, which is a key feature needed to communicate between Theorema and Java and which is also one of the main motivations to choose Java as the target language of the compiler (see Section 2.1).

The typical mathematical work consists of three general activities: proving, computing, and solving. Theorema supports all of them and thereby becomes, together with the extremely flexible and highly configurable front-end of Mathematica, a convenient environment for the entire mathematical exploration process. Although Theorema puts a special emphasis on proving, its computational capabilities are also a very important aspect, because whenever you implemented an algorithm, you want, of course, to try it out on some sample data. It is the main focus of this thesis to improve these computing capabilities of the current Theorema system.

In order to support proving, computing, and solving, Theorema comes up with its own language, which is, in fact, a version of higher order predicate logic without extensionality (see [Buch96a], [Buch99b]) and, therefore, is built-up of the following objects: constants, variables, terms, formulae, and quantifiers ([EFT92]). So, these ingredients form the core of Theorema's language, and they become of central interest when a definition stated in this language is translated into Java code. We will neither give a formal specification of the Theorema language, nor a more detailed description of it, but refer to a more general and detailed characterization of the whole Theorema system and its philosophy in [Tma97], [Buch98c], [Buch99a], [Tma99], [Tma00a], [Tma00b], and [Wind01].

1.2 Computing in Theorema

Theorema offers two different modes for computing: the *standard session* and the *computational session*. The standard session, which is the default mode when the Theorema system is started, offers the user command **`Compute`** for computing. A call to it has the following form:

```
Compute[Expression, using → KnowledgeBase]
```

For instance, to compute 18 + 7 in Theorema, you enter:

```
Compute[18 + 7, using → ⟨Built-in["Numbers"]⟩]
```

```
25
```

In this example the knowledge base only contains the package **`Built-in["Numbers"]`**, which is a built-in package of Theorema and contains several rewrite rules for natural numbers. As a more involved example, you may compute the set of all twin primes that are less than 100:

```
Compute[{⟨i, i + 2⟩      |      IsPrime[i] ⋀ IsPrime[i + 2]},
                   i=1,..,100
 using → ⟨Built-in["Numbers"],
   Built-in["Quantifiers"], Built-in["Connectives"]⟩]
```

```
{⟨3, 5⟩, ⟨5, 7⟩, ⟨11, 13⟩, ⟨17, 19⟩, ⟨29, 31⟩, ⟨41, 43⟩, ⟨59, 61⟩, ⟨71, 73⟩}
```

This time, the knowledge base contains the packages **`Built-in["Numbers"]`**, **`Built-in["Quantifiers"]`** (containing rewrite rules for Theorema's quantifiers, like the SetOf quantifier `{ | }`), and **`Built-in["Connectives"]`** (containing rewrite rules for logical connectives, like ∧).

As a last example, which also shows the flexibility of Theorema and the beauty of its syntax, we compute the set of perfect numbers that are less than or equal to 500. A perfect number is a positive integer which is the sum of its proper positive divisors.

$$\texttt{Compute}\left[\left\{\texttt{i}\ \underset{\texttt{i=1,..,500}}{\mid}\ \left(\sum_{\texttt{k}\in\left\{\texttt{j}\ \underset{\texttt{j=1,..,i-1}}{\mid}\ \texttt{j} \mid \texttt{i}\right\}}\texttt{k} = \texttt{i}\right)\right\},\right.$$
$$\left.\texttt{using} \to \langle\texttt{Built-in["Numbers"]},\ \texttt{Built-in["Sets"]},\ \texttt{Built-in["Quantifiers"]}\rangle\right]$$

```
{6, 28, 496}
```

In contrast to the standard session, computing in a computational session in Theorema works similar to working in Mathematica itself. In this computational mode you can simply enter the expression whose value you want to compute and do not need to put it into a **`Compute`** call. The Theorema user language provides the command **`ComputationalSession[]`** to enter a computational session and the command **`EndComputationalSession[]`** to leave it again. All calls between these two commands are directly interpreted and computed by Theorema. Furthermore, the philosophy of the computational session is that a knowledge base is built-up step by step by giving definitions or by importing environments that have been previously defined in the standard session ([Wind01]).

To execute the computations from above also in a computational session, we first have to tell Theorema which knowledge base we want to use :

```
Use[⟨Built-in["Numbers"], Built-in["Sets"],
  Built-in["Quantifiers"], Built-in["Connectives"]⟩]
```

Then, we can enter the computational session:

```
ComputationalSession[]
```

Theorema automatically imports the knowledge that we declared with the above **`Use`** command; from now on, all expressions that we enter are handled by Theorema using this knowledge:

$$\left\{\langle\texttt{i},\ \texttt{i}+\texttt{2}\rangle\ \underset{\texttt{i=1,..,100}}{\mid}\ \texttt{IsPrime[i]} \wedge \texttt{IsPrime[i + 2]}\right\}$$

```
{⟨3, 5⟩, ⟨5, 7⟩, ⟨11, 13⟩, ⟨17, 19⟩, ⟨29, 31⟩, ⟨41, 43⟩, ⟨59, 61⟩, ⟨71, 73⟩}
```

$$\left\{ \mathbf{i} \underset{\mathbf{i=1,\ldots,500}}{|} \left(\left(\sum_{\mathbf{k} \in \left\{ \mathbf{j} \underset{\mathbf{j=1,\ldots,i-1}}{|} \mathbf{j} \mid \mathbf{i} \right\}} \mathbf{k} \right) = \mathbf{i} \right) \right\}$$

```
{6, 28, 496}
```

Finally, we leave the computational session:

```
EndComputationalSession[]
```

For further details on both Theorema standard sessions and Theorema computational sessions we refer to [BuWi98] and [Wind99].

2 The Problem

Computing in Theorema's standard session and computational session (see Section 1.2) is rather slow, especially when dealing with big and nested data structures. So, in order to further improve the Theorema system and increase its versatility and usability, it was necessary to speed-up computations. This desire finally led to the implementation of the Theorema-Java Compiler, which is the main achievement of this thesis.

In the following chapters we will describe the key ideas and all details of the Theorema-Java Compiler. It is able to translate Theorema programs into equivalent Java byte code and, thereby, makes it possible to compute in Theorema tremendously faster than in Theorema's standard session and computational session. The road from an algorithm coded in Theorema's version of predicate logic to an equivalent Java program is long, rocky, and sometimes tricky, because a lot of obstacles have to be overcome. The difficulties in this translation basically arise from the inherent differences between the Theorema language and the Java language, and they lead, in particular, to the following challenges:

- Theorema is not a typed language, Java is. Expressions in Theorema do not have a specific type, whereas, on the other hand, Java is a strongly-typed programming language requiring all terms to have a defined type.

- Theorema supports higher order functions, Java does not. Theorema supports functions that take functions as parameters, whereas Java does not support methods that take methods as parameters (in fact, this is possible in Java by using its Reflection API, but for efficiency reasons we do not take this possibility into account). Nevertheless, the Theorema-Java Compiler does support higher order functions by applying a well known method to introduce such functions in Java: the method which should be passed as parameter is packed into a method of an object whose class implements a certain Java interface.

- Theorema supports sequence variables, Java does not. Sequence variables turn out to be extremely useful in practice since their use increases the elegance and the readability of programs. The current version of the Theorema-Java Compiler supports sequence variables at the very end of a pattern, like for instance in $f[x, y, \bar{z}]$ (sequence variables in Theorema are overbared, like $\bar{z}$). Although this is a limitation compared to the flexible support of sequence variables in Theorema, practice shows that this covers by far most of the cases.

- Theorema and Java are virtually two separated software systems. Nevertheless, it was necessary to connect them somehow in order to execute an algorithm on the Java side and transfer its result back to Theorema. In fact, Mathematica provides an interface to Java, which allows to instantiate Java objects and to call Java methods from within Mathematica. This interface, the so-called *J/Link*, was one of the reasons for choosing Java as the target language of the compiler.

In the following chapters of this part of the thesis we will give the main concepts of the Theorema-Java Compiler and also explain all its details. We will clearly and completely illustrate the sophisticated way the compiler combines the elegance of predicate logic, which is provided in the version of Theorema, and the efficiency of a modern, compiled programming language, namely Java. In Chapter 3 we will explain explicitly the three-steps procedure which is performed on every Theorema function in order to produce its equivalent Java byte code. Chapters 4, 5, 6, and 7 deal with the details of the translation of abstract data types, the translation of definitions with sequence variables, the translation of higher order functions, and the translation of functors, respectively.

After all these aspects of the translation are clear, we will describe in Chapter 8 how the user can run the compiled Java code from within Theorema. The remaining chapters of this part explain further details of the compiler, namely specific compiler settings and the organization in the file system of both the built-in Java files of the compiler and the files created by the user.

2.1 Why Did We Choose Java?

Java is an object-oriented, portable, and robust programming language originally developed by Sun Microsystems (www.sun.com). We chose it as the target language of the compiler for the following reasons:

- Java is nowadays a very popular language. It is modern and fully object-oriented and provides a huge library of auxiliary classes.
- Mathematica provides an interface to Java, the so-called J/Link. It provides a uniquely seamless interface to the Java environment and can be used in two ways:
  - Instantiate Java classes and call their methods from within Mathematica. The J/Link-library therefore provides Java classes (especially the class `com.wolfram.jlink.Expr`) to handle Mathematica expressions in Java.
  - Call Mathematica functions from within Java. This feature is not used by the compiler.
- The runtime performance of Java is really good, almost as good as C's.
- A Java compiler and the Java virtual machine are downloadable for free from the web-page of Sun, the inventor of Java.
- Java is platform independent, i.e., the Java virtual machine is available for Windows, Linux, and Mac OS.
- There is still an ongoing development of Java by Sun. From time to time a new version of the compiler is released including new and improved features.

2.2 A Short Summary of Features

The Theorema-Java Compiler comes up with several features and highlights:

- You may compile virtually any Theorema definition into executable Java code. Particularly, you may compile whole Theorema theories and definitions containing sequence variables (see Chapter 5) and functor definitions (see Chapter 7). This code runs much faster than computing within Theorema.
- You may compile Theorema definition once and use the compiled and fast Java program how many times you want.
- A special emphasis during the development of the compiler was put on the compilation of functors.
- The compilation to Java and the execution of compiled code is completely hidden from the user. That is, the user does not have to bother about the Java code and, actually, does not even come into contact with it at all. Nevertheless, the user is free to read the created Java code any time. Further information on how Java code is stored in the local file system is presented in Chapter 10.

2.3 System Requirements

The Theorema-Java Compiler requires an installed version of Theorema running on Mathematica 6 or higher and an installed Java Development Kit (JDK) 1.5 or higher.

All calculations and time measurements presented in this thesis were performed in Mathematica 6.0.0 and JDK 1.6.0_02 under Windows XP Home (Service Pack 2) on a Mobile DualCore Intel Pentium M with 1600 MHz and 2GB RAM.

2.4 A First Example

In this section we will show how the compiler is actually used to compile a simple Theorema theory and how to run the created Java code. In order to use Theorema and the Theorema-Java Compiler, you have to load the appropriate packages in Mathematica by the following commands:

```
Needs["Theorema`"]
Needs["Theorema`JavaCompiler`JavaCompiler`"]
```

In this example we define the function **`Merge`**, which merges two sorted lists such that the resulting list is again sorted, and a theory containing this definition:

```
Definition["Merge", any[x, x̄, y, ȳ],

     Merge[⟨x̄⟩, ⟨⟩] = ⟨x̄⟩
     Merge[⟨⟩, ⟨ȳ⟩] = ⟨ȳ⟩                                          ]
     Merge[⟨x, x̄⟩, ⟨y, ȳ⟩] = { x ⌣ Merge[⟨x̄⟩, ⟨y, ȳ⟩]  ⇐  x < y
                              { y ⌣ Merge[⟨x, x̄⟩, ⟨ȳ⟩]  ⇐  otherwise


Theory["MergeTheory",
    Definition["Merge"]]
```

We can use this theory to compute in Theorema:

```
Compute[Merge[⟨4, 20, 30⟩, ⟨1, 2, 5, 32⟩],
  using → ⟨Built-in["Tuples"], Built-in["Numbers"],
    Built-in["Connectives"], Theory["MergeTheory"]⟩] // AbsoluteTiming
```

```
{0.1250000, ⟨1, 2, 4, 5, 20, 30, 32⟩}
```

So, the result is **⟨1, 2, 4, 5, 20, 30, 32⟩**, and it took Theorema 0.125 seconds to compute it. Now, we may compile this theory using the Theorema-Java Compiler creating a fast Java program. For this, we provide the command **Java–Theory2Java**:

```
Java-Theory2Java[Theory["MergeTheory"]]
```

In order to access the Java program which the compiler just created and use it for computations, we first have to put the theory "MergeTheory" into the knowledge base of the Java-sided execution process:

```
Java-UseTheories[{"MergeTheory"}]
```

Finally, we can use the Java program for executing a computation:

```
Java-Compute[Merge[⟨4, 20, 30⟩, ⟨1, 2, 5, 32⟩]] // AbsoluteTiming
```

```
{0.0156250, ⟨1, 2, 4, 5, 20, 30, 32⟩}
```

The result is the same as above, but the compiled Java program needed just 0.0156 seconds to produce it.

3 The Three Steps of Translation

In this chapter we describe the three general steps to translate given Theorema definitions into executable Java byte code. These steps are performed in all cases, no matter whether a single definition, a whole theory including several definitions, or a functor is compiled. The first and the last step are quite simple to perform, whereas the second step is more involved since it includes quite challenging steps, for instance, translating sequence variables and higher order function.

In the course of translating a given Theorema function (or functor), its Theorema definition is first transcribed into an intermediate format rid of pattern matching. In the second step, which is also the core step in the whole three-stage translation, each function definition, given in the intermediate format, is translated into (well readable) Java source code. The basics of this step are described in Section 3.2. All the details on how to translate abstract data types, definitions including sequence variables, higher order function, and functors are described in full detail in the chapters 4, 5, 6, 7, respectively. In the third step, the Java source code is compiled to byte code using a conventional Java compiler. These three steps are always performed and, thereby, form the general flow of translation, which is depicted in Figure 3.1.

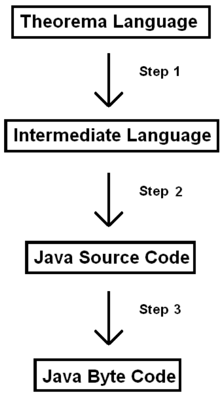

Figure 3.1: Flow of Translation

3.1 Eliminating Pattern Matching

Pattern matching is a very powerful and flexible tool to process data based on its structure. Its real power comes from matching patterns and accordingly bind variables at the same time. Together with conditional execution constructs, pattern matching leads to a very elegant and structured way of programming. Both Mathematica and Theorema support defining functions using these mechanisms, and the following example shows how they can be used in Theorema: The function **`Ind-Plus`** adds two natural numbers that are represented by the following data structure: **`0`** is represented by the constant **`Zero`**, **`1`** is represented by **`Succ[Zero]`**, **`2`** by **`Succ[Succ[Zero]]`**, **`3`** by **`Succ[Succ[Succ[Zero]]]`**, and so on.

```
Definition["InductivePlus", any[x, y],
    Ind-Plus[x, Zero] = x
    Ind-Plus[x, Succ[y]] = Succ[Ind-Plus[x, y]]]
```

The function distinguishes two cases that are detected by pattern matching. The first rewrite rule

```
Ind-Plus[x, Zero] = x
```

only matches if the second parameter is equal to the constant `Zero`. The second rewrite rule

```
Ind-Plus[x, Succ[y]] = Succ[Ind-Plus[x, y]]
```

only matches if the head of the second parameter is equal to the constant `Succ`.

In order to add the natural numbers **2** and **3**, you compute:

```
Compute[Ind-Plus[Succ[Succ[Zero]], Succ[Succ[Succ[Zero]]]],
 using → ⟨Definition["InductivePlus"]⟩]
```

```
Succ[Succ[Succ[Succ[Succ[Zero]]]]]
```

Already this simple example demonstrates the increased elegance and readability of programs which use pattern matting for their definition. However, Java does not support such mechanisms, and, hence, the first step on the way of translating a Theorema definition into Java code is always to eliminate pattern matching. For that, the Theorema-Java Compiler translates the Theorema definitions into an intermediate language rid of pattern matching. Both the intermediate language and the Mathematica package which does this elimination were originally developed by Tudor Jebelean ([Jebe07]) and later adapted by the author. The following line shows how the function `Ind-Plus` from above can be defined in Mathematica without pattern matching:

```
Ind-Plus[x_, y_] :=
  If[y === Zero, x, If[Head[y] === Succ, Succ[Ind-Plus[x, y[[1]]]]]];
```

Instead of defining two cases which are distinguished by the pattern of the second parameter of the function, this definition uses an `If`-clause and explicitly checks the structure of `y` by evaluating `y===Zero` and `Head[y]===Succ`. You may again compute **2** plus **3**:

```
Ind-Plus[Succ[Succ[Zero]], Succ[Succ[Succ[Zero]]]]
```

```
Succ[Succ[Succ[Succ[Succ[Zero]]]]]
```

The function `Ind-Plus` coded in the above mentioned intermediate language is:

```
•DeFun[•sig["Ind-Plus"], ⟨_param1, _param2⟩,
 •Conditional[⟨⟨•const[Zero] = _param2, _param1⟩,
   ⟨•const[Succ] = •Head[_param2], •Expr[•const[Succ],
     ⟨•Expr[•const[Ind-Plus], ⟨_param1, •Arg[1, _param2]⟩]⟩]⟩⟩]]
```

This expression consists of 4 parts: The head **`•DeFun`** indicates that this is a function definition; **`•DePre`** would indicate a predicate definition. The first part, **`•sig["Ind-Plus"]`**, states the name of this function, and the second part, `〈_param1, _param2〉`, states the parameter list. The third part, **`•Conditional[…]`**, defines the body of the function and is a list of condition-expression pairs; if a condition holds, the corresponding expression is returned. Accordingly, in the case of a predicate definition, the body is a list of condition-condition pairs.

In the remaining part of this section we will explain the elimination of pattern matching in general by considering a unary function. A function of arity n can be translated by iteratively applying the transcription presented below.

The left hand side of the definition of a unary function ***f*** may basically have either the form `f[c]` where ***c*** is a constant symbol, either the form `f[x]` where ***x*** is a variable, or the form `f[C[x1, ..., xn]]`, where ***C*** is constructor (see Chapter 4) of arity n and x_i (for $1 \le i \le n$) is either a constant symbol, a variable, or again a nested expression of the form `D[ ...]` where **D** is a constructor. Note that a more flexible shape of function definitions is possible if sequence variables are used, see Chapter 5.

The general shape of ***f*** coded in the intermediate language looks like this:

```
•DeFun[•sig["f"], 〈_param1〉, •Conditional[〈〈Condition, Expression〉〉]]
```

where ***Condition*** and ***Expression*** are a condition and an expression, respectively, depending on ***f***. As stated above, we have to distinguish three cases: If **f** has the form `f[c] = e` (where **e** is some expression), ***Condition*** has the following value

```
〈•const[c] = _param1, e*〉
```

where ***e**** is the translation of ***e*** into the intermediate format. If **f** is of the form `f[x] = e`, ***Condition*** is

```
〈true, e*〉
```

because there is no condition on the parameter, and ***e**** is again the translation of ***e*** into the intermediate format. If **f** is of the third shape, `f[C[x1, .., xn]] = e`, the ***Condition*** looks like this:

```
〈(•const[C] = •Head[_param1]) ⋀ T, e*〉
```

e* is obtained by translating **e** into the intermediate format and replacing x_i by `•Arg[i, _param1]` (for all $1 \le i \le n$), a construct of the intermediate language which accesses the i-th component of *_param1*. ***T*** is obtained by applying this three-folded case distinction recursively to x_1, ..., x_n and forming the conjunction. To address x_i and formulate a condition on it, it is also replaced by `•Arg[i, _param1]`.

For example, the function definition `f[C[x, D[y], E[a]]] = e` (where **C**, **D**, and **E** are constructors, **a** is a constant symbol, and **x** and **y** are variables) is translated into the function body

```
〈((•const[C] = •Head[_param1]) ⋀ (•const[D] = •Head[•Arg[2, _param1]]) ⋀
  (•const[E] = •Head[•Arg[3, _param1]]) ⋀
  (•const[a] = •Arg[3, •Arg[1, _param1]]), e*〉
```

where ***e**** is obtained by translating **e** into the intermediate format and replacing **x** and **y** by **•Arg[1,** *_param1*] and **•Arg[2, •Arg[2,** *_param1*]], respectively.

This translation is accordingly applied to predicate definitions.

3.2 Creating Java Source Code

The second step on the way of translating a Theorema definition into executable Java code is to turn definitions in intermediate format into actual Java source code.

Each function and each predicate (given in intermediate format) is parsed, and all occurring conditions and expressions are translated into Java code. Of course, this process of translation is a recursive procedure since each condition and each expression may again contain conditions and expressions. In other words, the definition of a function, which is made up of nested conditions and expressions, is recursively translated into Java code.

Also, since Theorema and Java have different naming conventions (e.g., Theorema allows dashes in names, Java does not), a renaming of identifiers (variables, class names, etc.) has to be performed: All appearing language keywords of Java (e.g., `while`, `for`, `new`) are changed by prepending and appending an underscore, and dashes are replaced by underscores, and blanks are eliminated.

The real challenge of this step is the translation of language constructs in Theorema which do not exist in Java, for instance, sequence variables, higher order functions, quantifiers, and functors. To see how the translation of a simple function works, let us take the example from the previous section and have a look at the body of the function **Ind-Plus** in intermediate format:

```
•Conditional[〈〈•const[Zero] = _param2, _param1〉,
  〈•const[Succ] = •Head[_param2], •Expr[•const[Succ],
    〈•Expr[•const[Ind-Plus], 〈_param1, •Arg[1, _param2]〉]〉]〉〉]
```

The head of this body is **•Conditional**, hence, we know that we have to create a branching statement (`if`-clause). Since we are about to define a function (**•DeFun**), the **•Conditional**-expression contains a tuple of pairs of one condition and one expression, whereas in the case of defining a predicate it would contain a tuple of pairs of two conditions. In this example the first pair is

```
〈•const[Zero] = _param2, _param1〉
```

So, we translate the first entry of the tuple into the Java condition (see also the translation of abstract data types in Chapter 4):

```
(((ExtendedData)_param2).isZero())
```

The second entry of the tuple, i.e., the value which is returned by **Ind-Plus** if the condition in the first tuple entry is fulfilled, is trivially translated into the Java expression

```
_param1
```

In the same way, the second pair

```
〈•const[Succ] = •Head[_param2], •Expr[•const[Succ],
  〈•Expr[•const[Ind-Plus], 〈_param1, •Arg[1, _param2]〉]〉]〉
```

is translated into Java code. The translation of the condition is

```
(((ExtendedData)_param2).isSucc())
```

The translation of the expression is

```
new Succ(ind_Plus(_param1,_param2.arg(1)))
```

Finally, the two conditions and the two expressions are put together according to the rules of **•Conditional**. So, the Java code of the function **Ind–Plus** is:

```
Data ind_Plus(Data _param1,Data _param2)
{
    if ((((ExtendedData)_param2).isZero()))
    {
        return _param1;
    }//if

    if ((((ExtendedData)_param2).isSucc()))
    {
        return new Succ(ind_Plus(_param1,_param2.arg(1)));
    }//if
}//ind_Plus
```

In general, the translation of functions and predicates from intermediate format into Java code consists of three major parts: the translation of conditions, the translation of expressions, and the translation of conditional branchings. Compiling conditions comprises the translation of truth values, logical functions (**AND**, **OR**, **NOT**), equalities, the ∀ bounded quantifier, and the ∃ bounded quantifier. Compiling expressions comprises the translation of tuples, the TupleOf quantifier (〈 | 〉), the SetOf quantifier ({ | }), the ∑ quantifier, the SuchThat quantifier (æ), sequence variables, and constants. Many of these translations involve technical details, like dealing with sequence variables (see Chapter 5), or calling a function of a certain domain (see Chapter 7). Moreover, special efforts have to be made to translate the bounded quantifier constructs, i.e., TupleOf, SetOf, ∀, ∃, ∑, and SuchThat, into correct Java code. The created Java code of a translated quantifier is packed into an auxiliary method named "aux*n*" (where *n* is 1,2,3,…), and this method is called with the appropriate parameters.

In the following sections we will describe in detail the translation of each quantifier into Java source code. For this, for each quantifier, we will define a function **f** in Theorema-like syntax using a general form of the quantifier and show its equivalent on the Java side, written in a Java-like syntax.

3.2.1 The TupleOf Quantifier

Theorema supports two types of this quantifier, which differ in the range the index variable runs over. The first type uses a so-called *integer range* and has the following form:

$$f[x_1, \ldots, x_n] = \left\langle g[i, x_1, \ldots, x_n] \underset{i=h[x_1,\ldots,x_n],\ldots,k[x_1,\ldots,x_n]}{|} c[i, x_1, \ldots, x_n] \right\rangle$$

The index variable i runs from the value $h[x_1, \ldots, x_n]$ to the value $k[x_1, \ldots, x_n]$; if the condition $c[i, x_1, \ldots, x_n]$ holds, the element $g[i, x_1, \ldots, x_n]$ becomes part of the generated tuple. The compiler translates this function definition into the following code, which is given here in a Java-like syntax:

```
Data f(Data param_1,...,Data param_n)
{
    return aux1(param_1,...,param_n,h(param_1,...,param_n),k(param_1,...,param_n));
}//f

Tuple aux1(Data param_1,...,Data param_n,int auxvar1,int auxvar2)
{
    Data[] auxvar3 = new Data[auxvar2-auxvar1+1];
    int auxvar4 = 0;
    for(int i=auxvar1;i≤auxvar_2;i++)
        if (c(i,param_1,...,param_n))
        {
            auxvar3[auxvar4] = g(i,param_1,...,param_n);
            auxvar4++;
        }//if
    return new Tuple(auxvar3,auxvar4);
}//aux1
```

So, the TupleOf quantifier with an integer range is translated into a separate auxiliary method which essentially consists of a `for` loop.

The second type of the TupleOf quantifier uses a so-called set range and has the following form:

$$f[x_1, \ldots, x_n, S] = \left\langle g[i, x_1, \ldots, x_n] \underset{i\in S}{|} c[i, x_1, \ldots, x_n] \right\rangle$$

The index variable i runs through the values of the set S; whenever the condition $c[i, x_1, \ldots, x_n]$ holds, the element $g[i, x_1, \ldots, x_n]$ becomes part of the generated tuple. The compiler translates this function definition into the following code, which is given again in a Java-like syntax:

```
Data f(Data param_1,...,Data param_n,Data S)
{
    return aux1(param_1,...,param_n,S);
}//f

Tuple aux1(Data param_1,...,Data param_n,Set s)
{
    int auxvar3 = 0;
    int auxvar4 = s.size();
    Data i;
    Data[] auxvar1 = new Data[auxvar4];

    for(int auxvar5=0;auxvar5<auxvar4;auxvar5++)
    {
        i = s.arg(auxvar5+1);
        if (c(i,param_1,...,param_n))
        {
            auxvar1[auxvar3] = g(i,param_1,...,param_n);
            auxvar3++;
        }//if
    }//for
    return new Tuple(auxvar1,auxvar3);
}//aux1
```

So, the TupleOf quantifier with a set range is translated into a separate auxiliary method which essentially consists of a `for` loop.

3.2.2 The SetOf Quantifier

The SetOf quantifier is very similar to the TupleOf quantifier: instead of square brackets it uses curly brackets, and instead of the Java type `Tuple` it uses `Set`.

3.2.3 The $\sum$ Quantifier

The $\sum$ quantifier also comes in two forms: one with an integer range, one with a set range. The first one has the following shape:

$$f[x_1, \dots, x_n] = \sum_{\substack{i=h[x_1,\dots,x_n],\dots,k[x_1,\dots,x_n] \\ c[i,x_1,\dots,x_n]}} g[i, x_1, \dots, x_n]$$

It is translated into the following code:

```
Data f(Data param_1,…,Data param_n)
{
    return aux1(param_1,…,param_n,h(param_1,…,param_n),k(param_1,…,param_n));
}//f

BI_Number aux1(Data param_1,…,Data param_n,int auxvar1,int auxvar2)
{
    BI_Rational auxvar3 = BI_Rational.ZERO;

    for(int i=auxvar1;i≤auxvar2;i++)
        if (c(i,param_1,…,param_n))
            auxvar3 = g(i,param_1,…,param_n);
    return auxvar3;
}//aux1
```

The second type of the ∑ quantifier, which uses a set range, has the following form:

$$f[x_1, \ldots, x_n, S] = \sum_{\substack{i \in S \\ c[i, x_1, \ldots, x_n]}} g[i, x_1, \ldots, x_n]$$

Its corresponding Java code is:

```
Data f(Data param_1,…,Data param_n,Data S)
{
    return aux1(param_1,…,param_n,S);
}//f

BI_Number aux1(Data param_1,…,Data param_n,Set s)
{
    int auxvar3 = s.size();
    Data i;
    BI_Rational auxvar1 = BI_Rational.ZERO;

    for(int auxvar4=0;auxvar4<auxvar3;auxvar4++)
    {
        i = s.arg(auxvar4+1);
        if (c(i,param_1,…,param_n))
            auxvar1 = (BI_Rational)auxvar1.add(g(i,param_1,…,param_n));
    }//for
    return auxvar1;
}//aux1
```

3.2.4 The SuchThat Quantifier

The SuchThat quantifier(æ) is used in explicit definitions of new function symbols, where it actually is only used as an abbreviation for an implicit definition of the new symbol. For example,

$$\underset{x}{\forall}\left(f[x] = \underset{y}{æ}\left(y^2 = x\right)\right)$$

is considered as an abbreviation of the formula

$$\underset{x}{\forall}\left(f[x]^2 = x\right)$$

Also the SuchThat quantifier comes in two shapes, the one with the integer range looks like this:

$$f[x_1, \ldots, x_n] = \underset{\substack{i=h[x_1,\ldots,x_n],\ldots,k[x_1,\ldots,x_n] \\ c[i,x_1,\ldots,x_n]}}{\text{æ}} g[i, x_1, \ldots, x_n]$$

It is translated into the following Java code:

```
Data f(Data param_1,…,Data param_n)
{
    return aux1(param_1,…,param_n,h(param_1,…,param_n),k(param_1,…,param_n));
}//f

Data aux1(Data param_1,…,Data param_n,int auxvar1,int auxvar2)
{
    for(int i=auxvar1;i≤auxvar2;i++)
        if(c(i,x_1,…,x_n) && g(i,param_1,…,param_n))
            return BI_Integer.valueOf(i);
    return null;
}//aux1
```

The second one, which uses a set range, has the following form:

$$f[x_1, \ldots, x_n, S] = \underset{\substack{i \in S \\ c[i,x_1,\ldots,x_n]}}{\text{æ}} g[i, x_1, \ldots, x_n]$$

It is translated into:

```
Data f(Data param_1,…,Data param_n,Data S)
{
    return aux1(param_1,…,param_n,S);
}//f

Data aux1(Data param_1,…,Data param_n,Set s)
{
    int auxvar1 = S.size();
    Data i;

    for(int auxvar2=0;auxvar2<auxvar1;auxvar_2++)
    {
        i = s.arg(auxvar3+1);
        if(c(i,x_1,…,x_n) && g(i,param_1,…,param_n))
            return i;
    }//for
    return null;
}//aux1
```

3.2.5 The ∀ Quantifier

A typical usage of the ∀ quantifier using an integer range looks like this:

$$p[x_1, \ldots, x_n] \Leftrightarrow \left(\underset{\substack{i=h[x_1,\ldots,x_n],\ldots,k[x_1,\ldots,x_n] \\ c[i,x_1,\ldots,x_n]}}{\forall} g[i, x_1, \ldots, x_n] \right)$$

The Theorema-Java Compiler translates this predicate into the following code:

```
BooleanData p(Data param1,...,Data paramn)
{
    return convertBooleanToData(aux1(param1,...,paramn,h(param1,...,paramn),
        k(param1,...,paramn)));
}//p

boolean aux1(Data param1,...,Data paramn,int auxvar1,int auxvar2)
{
    for(int i=auxvar1;i≤auxvar2;i++)
        if(c(i,x1,...,xn) && !g(i,param1,...,paramn))
            return false;
    return true;
}//aux1
```

Using a set range, the ∀ quantifier typically comes in this shape:

$$p[x_1, \ldots, x_n, S] \Leftrightarrow \left(\underset{\substack{i\in S \\ c[i,x_1,\ldots,x_n]}}{\forall} g[i, x_1, \ldots, x_n] \right)$$

The translated code in the Java-like syntax is:

```
BooleanData p(Data param_1,... ,Data param_n,Data S)
{
    return convertBooleanToData(aux1(param_1,... ,param_n,S));
}//p

boolean aux1(Data param_1,... ,Data param_n,Set s)
{
    int auxvar1 = s.size();
    Data i;

    for(int auxvar2=0;auxvar2<auxvar1;auxvar2++)
    {
        i = s.arg(auxvar2+1);
        if(c(i,x_1,... ,x_n) && !g(i,param_1,... ,param_n))
            return false;
    }//for
    return true;
}//aux1
```

3.2.6 The ∃ Quantifier

The ∃ quantifier also comes in two forms: one with an integer range, one with a set range. The first one has the following shape:

$$p[x_1, \ldots, x_n] \Leftrightarrow \left(\underset{\substack{i=h[x_1,\ldots,x_n],\ldots,k[x_1,\ldots,x_n]\\ c[i,x_1,\ldots,x_n]}}{\exists} g[i, x_1, \ldots, x_n] \right)$$

The corresponding code is:

```
BooleanData p(Data param_1,... ,Data param_n)
{
    return convertBooleanToData(aux1(param_1,... ,param_n,h(param_1,... ,param_n),
        k(param_1,... ,param_n)));
}//p

boolean aux1(Data param_1,... ,Data param_n,int auxvar1,int auxvar2)
{
    for(int i=auxvar1;i≤auxvar2;i++)
        if(c(i,x_1,... ,x_n) && g(i,param_1,... ,param_n))
            return true;
    return false;
}//aux1
```

The second form is:

$$p[x_1, \ldots, x_n, S] \Leftrightarrow \left(\underset{\substack{i\in S\\ c[i,x_1,\ldots,x_n]}}{\exists} g[i, x_1, \ldots, x_n] \right)$$

Its corresponding code is:

```
BooleanData p(Data param_1,… ,Data param_n,Data S)
{
    return convertBooleanToData(aux1(param_1,… ,param_n,S));
}//p

boolean aux1(Data param_1,… ,Data param_n,Set s)
{
    int auxvar1 = s.size();
    Data i;

    for(int auxvar2=0;auxvar2<auxvar1;auxvar2++)
    {
        i = s.arg(auxvar2+1);
        if(c(i,x_1,… ,x_n) && g(i,param_1,… ,param_n))
            return true;
    }//for
    return false;
}//aux1
```

3.3 Creating Java Byte Code

The final step of the translation from Theorema to Java is the compilation of the created Java source code to Java byte code. This can be done by any available Java compiler which supports the JDK 1.5 or newer. We recommend to use Sun's compiler since we did all the tests with this one (Sun's JDK 1.6). The user must assure that the Java compiler (`javac`) is accessible, i.e., the system environment variable `PATH` has to be set accordingly. After the Java source files were successfully created, the Theorema-Java Compiler automatically calls `javac` to compile all produced source files to Java byte code.

4 Translation of Abstract Data Types

The Theorema-Java Compiler supports the translation of abstract data types that are defined in Theorema into Java code. In this part of the thesis we show first a simple example and then explain the general way of the translation in full detail. The original ideas and examples presented in this chapter were mainly given by Martin Giese ([Gies07]).

4.1 An Example: `Plus`

This example is quite similar to the one in Section 3.1; it just uses a more natural notation. Again, we define the integers inductively: **0** is represented by the constant Z, **1** is represented by Z^{+}, **2** by Z^{++}, **3** by Z^{+++}, etc. Please note that an expression of the form T^{+} is internally stored as `SuperPlus[T]`, for every expression T. Similarly, an expression of the form $T+S$ (for expression S and T) is internally stored as `Plus[S, T]`. For instance, the addition **2+3** is represented by the expression $Z^{++}+Z^{+++}$, which is internally stored as `Plus[SuperPlus[SuperPlus[Z]], SuperPlus[SuperPlus[SuperPlus[Z]]]]`.

Here is the definition of the function `Plus` and the associated theory:

```
Definition["Plus", any[x, y],
     x + Z = x
     x + y⁺ = (x + y)⁺ ]

Theory["Plus",
    Definition["Plus"]]
```

If we, for instance, want to add **2** and **3**, we simply compute

```
Compute[Z⁺⁺ + Z⁺⁺⁺, using → ⟨Theory["Plus"]⟩]
```

$$((((Z^{+})^{+})^{+})^{+})^{+}$$

To compile this Theorema theory to Java we have to enter

```
Java-Theory2Java[Theory["Plus"]]
```

In the second step of the general, three-stage flow of translation (see Chapter 3) the compiler has to create a Java-sided representation of the constant symbol Z and of the unary symbol `SuperPlus`, which is the internal representation of $^{+}$. The symbols Z and `SuperPlus` are called *constructors*, which are generally represented by Java classes. What can these classes look like? At this point, typecasting plays a crucial rôle. In the current design of the Theorema-Java Compiler the type of all parameters of all user defined functions has to be `Data`, which is an abstract class provided by the framework of the Theorema-Java Compiler, see Section 10.2.1. Since Z is a possible parameter of the function `Plus`, the representation of the constant ***Z*** on the Java side has to be a class that is a subclass of `Data`. Accordingly, the representation of **`SuperPlus`** is a class that is also a subclass of `Data` and has a constructor taking one parameter of type `Data`. Actually, these representation classes are not direct subclasses of `Data`, but are derived from the intermediate, abstract class `ExtendedData`, which is directly subclassing `Data` and also created automatically by the compiler.

Figure 4.1 shows the UML class diagram of these classes. In this figure, the `Data` class is colored in grey since it is not created automatically in the flow of translation, but provided by the framework of the Theorema-Java Compiler. The three other classes, which are created completely automatically by the compiler, are colored in black.

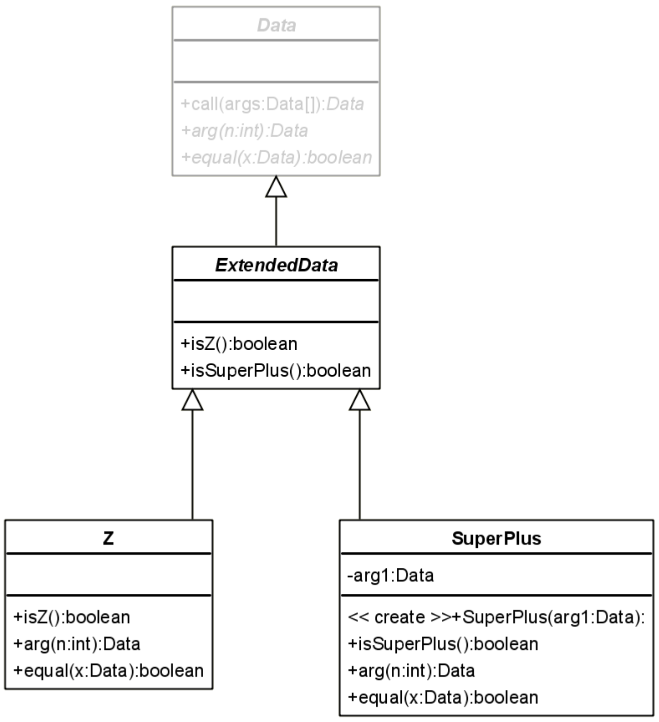

Figure 4.1: UML Diagram of the Classes of the Theory "Plus"

We will now present the actual implementation of the representation classes, according to the class model in Figure 4.1. The first Java class which is created is `ExtendedData`. It is an abstract class and a direct subclass of `Data`. Moreover, it contains identifying methods for its two possible implementations, namely the class `Z` and the class `SuperPlus`. That is, it contains boolean, non-abstract functions `isZ` and `isSuperPlus`, which both return `false` in their implementation in `ExtendedData`.

```
public abstract class ExtendedData extends Data
{
    public boolean isZ()
    {
        return false;
    }//isZ

    public boolean isSuperPlus()
    {
        return false;
    }//isSuperPlus
}//class ExtendedData
```

The class `Z` is a direct subclass of `ExtendedData` and overloads the function `isZ`, which returns `true` in the class `Z`. Furthermore, it implements the function `equal` (and also others which are omitted here for the sake of simplicity), which is (are) inherited from `Data`.

```
public class Z extends ExtendedData
{
    public boolean isZ()
    {
        return true;
    }//isZ

    public boolean equal(Data x)
    {
        if (x instanceof ExtendedData)
        {
            return (((ExtendedData)x).isZ());
        }//if
        else
        {
            return false;
        }//else
    }//equal

    ...

}//class Z
```

The class `SuperPlus` is also a direct subclass of `ExtendedData` but overloads the function `isSuperPlus`, which returns `true` in the class `SuperPlus`. Like the class `Z`, it has to implement the function `equal` (and also others which are again omitted here for the sake of simplicity), because it is (are) inherited from the abstract class `Data`.

```
public class SuperPlus extends ExtendedData
{
    private Data arg1;
    public SuperPlus(Data arg1)
    {
        this.arg1=arg1;
    }//SuperPlus

    public boolean isSuperPlus()
    {
        return true;
    }//isSuperPlus

    public Data arg(int n)
    {
        if (n==1)
        {
            return arg1;
        }//if

        return null;
    }//arg

    public boolean equal(Data x)
    {
        if (x instanceof ExtendedData)
        {
            return (((ExtendedData)x).isSuperPlus()&&
                arg(1).equal(((SuperPlus)x).arg(1)));
        }//if
        else
        {
            return false;
        }//else
    }//equal

    ...

}//class SuperPlus
```

The classes `Z` and `SuperPlus` can now be used to express the chosen data structure. Given the Java classes above, we can, for instance, create the object $Z^{+^{+^{+}}}$ (representing the natural number **3**) as a Java object:

```
new SuperPlus(new SuperPlus(new SuperPlus(new Zero())))
```

As the final step, we have to translate the algorithm `Plus` into Java source code. Generally, all algorithms of a theory are collected in the class `Algorithms`. As shown in Section 3.2, each rewrite rule in the Theorema definition of `Plus` is translated (via the intermediate language) into an `if`-clause on the Java side. The signature of the Java implementation of `Plus` is

```
Data plus(Data _param1,Data _param2)
```

The first rewrite rule in the definition of `Plus`

$$x + Z = x$$

is translated into

```
if (((ExtendedData)_param2).isZ()))
{
    return _param1;
}//if
```

Accordingly, the second rewrite rule

$$x + y^{+} = (x + y)^{+}$$

is translated into

```
if (((ExtendedData)_param2).isSuperPlus()))
{
    return new SuperPlus(plus(_param1,_param2.arg(1)));
}//if
```

Hence, the whole `Algorithms` class looks like this:

```
public class Algorithms
{
    public static Data plus(Data _param1,Data _param2)
    {
        if (((ExtendedData)_param2).isZ()))
        {
            return _param1;
        }//if

        if (((ExtendedData)_param2).isSuperPlus()))
        {
            return new SuperPlus(plus(_param1,_param2.arg(1)));
        }//if

        return null;
    }//plus
}//class Algorithms
```

This is the well structured and well readable Java source code which the Theorema-Java Compiler finally created and compiled to Java byte code. If we want to use it for computations, we first have to put the theory "Plus" into the knowledgebase of the Java-sided execution process:

```
Java-UseTheories[{"Plus"}]
```

We may now compute **2+3**:

```
Java-Compute[Z⁺⁺ + Z⁺⁺⁺]
```

$$((((Z^{+})^{+})^{+})^{+})^{+}$$

Before describing the details of the general aspects of this translation in the next section, we want to easily broaden the above example by adding another algorithm, namely multiplication. Thus, we define the multiplication in our inductive data structure:

```
Definition["Times", any[x, y],
     x * Z = Z
     x * y⁺ = x * y + x
     Z * y = Z
     x⁺ * y = y + x * y ]


Theory["Plus-Times",
     Definition["Plus"]
     Definition["Times"] ]
```

We may compute **(1+2)*3**:

```
Compute[(Z⁺ + Z⁺⁺) * Z⁺⁺⁺, using → ⟨Theory["Plus-Times"]⟩]
```

$$(((((((((Z^{+})^{+})^{+})^{+})^{+})^{+})^{+})^{+})^{+}$$

The result is, as expected, **9**. Let us compile the theory and have a look at its corresponding Java source code:

```
Java-Theory2Java[Theory["Plus-Times"]]
```

The thereby created class `Algorithms` is identical to the one shown above except that it has an additional method `times`:

```
public static Data times(Data _param1,Data _param2)
{
    if (((ExtendedData)_param2).isZ())
    {
        return ExtendedFactory.getZ();
    }//if

    if (((ExtendedData)_param2).isSuperPlus())
    {
        return plus(times(_param1,_param2.arg(1)),_param1);
    }//if

    if (((ExtendedData)_param1).isZ())
    {
        return ExtendedFactory.getZ();
    }//if

    if (((ExtendedData)_param1).isSuperPlus())
    {
        return plus(_param2,times(_param1.arg(1),_param2));
    }//if

    return null;
}//times
```

Again, we may now compute using the compiled Java code:

```
Java-UseTheories[{"Plus-Times"}]
```

Java-Compute$\left[\left(Z^{+}+Z^{++}\right)*Z^{+++}\right]$

$$\left(\left(\left(\left(\left(\left(\left(\left(Z^{+}\right)^{+}\right)^{+}\right)^{+}\right)^{+}\right)^{+}\right)^{+}\right)^{+}\right)^{+}$$

4.2 General Translation

In the previous section we exemplarily presented Theorema programs which work on a data structure that is built by so-called constructor terms, and we showed its corresponding Java source code. We will now explain the translation of such Theorema programs into Java source code in full generality.

The idea for the current way of translation of such programs was given by Martin Giese, [Gies07], and is a general concept which associates to each type of constructor term a Java class. A *constructor* is a constant symbol with a certain arity and a name that is different from all the names of algorithms in the current knowledge base. Please note that also the angle brackets (⟨…⟩) can be viewed as constructors, but with an arbitrary arity. A common example of a data structure using constructors are Lisp-style lists: Given the 0-ary constructor **nil** and the binary constructor **cons**, we can easily construct lists of arbitrary length, e.g., the term **cons[18,cons[0,cons[7,nil]]]** is the representation of the list **⟨18,0,7⟩**. Another example is the representation of multivariate polynomials: Given the binary constructor **`Mon`** and the *n*-arity constructor **`PP`**, we can represent polynomials in *n* variables. For instance, the polynomial $7\,x^2\,y\,z^3 - 2\,y\,z + 5\,z$ can be represented by the tuple **`⟨Mon[7,PP[2,1,3]],Mon[-2,PP[0,1,1]],Mon[5,PP[0,0,1]]⟩`**.

Given a Theorema theory, we can extract both the occurring constructors and the defined algorithms. Hence, in the general, three-part flow of translation (see Chapter 3), the compiler now has to do more in Step 2: beside translating the algorithms into Java source code, it has to create a representation of all occurring constructors. In the course of this these constructors are divided into two parts: the 0-ary constructors $C_{0,1}, \ldots, C_{0,n_0}$ and the constructors $C_{i,1}, \ldots, C_{i,n_i}$ of arity i and $i > 0$. In the current implementation of the compiler each constructor is represented by a Java class, which is automatically created when the associated Theorema theory is compiled. All these Java classes are derived from the intermediate class `ExtendedData`, which is abstract and directly derived from `Data` (provided by the framework of the Theorema-Java Compiler, see Section 10.2.1). `ExtendedData` is also created automatically and contains identifying methods for each constructor class.

Figure 4.2 shows the over-all class design as an UML class diagram. For the sake of clarity, for each of the two types of constructors, namely the 0-ary ones and the non-zero-ary ones, only one representative class is depicted: the class $C_{0,j}$ stands for a 0-ary constructor, the class $C_{i,j}$ stands for a constructor of arity i with $i > 0$. Furthermore, the `Data` class is colored in grey to indicate that it is not created automatically in the flow of translation, but provided by the framework of the Theorema-Java Compiler. The figure schematically shows several properties of the automatically generated classes:

- For every occurring constructor the class `ExtendedData` contains an identifying boolean method `isC`$_{i,j}$ ($i \geq 0$, $1 \leq j \leq n_i$) which yields `false`.

- The class $C_{0,j}$, representing a constructor of arity zero, overwrites `ExtendedData`'s method `isC`$_{0,j}$ by a method returning `true`. By necessity, the class $C_{0,j}$ also implements the methods `arg` and `equal`, which are inherited from the abstract class `Data`.

- The class $C_{i,j}$, representing a constructor of arity i ($i > 0$), overwrites `ExtendedData`'s method `isC`$_{i,j}$ by a method returning `true`. By necessity, the class $C_{i,j}$ also implements the methods `arg` and `equal`, which are inherited from the abstract class `Data`. Furthermore, it contains i private fields of type `Data` and a constructor of arity i.

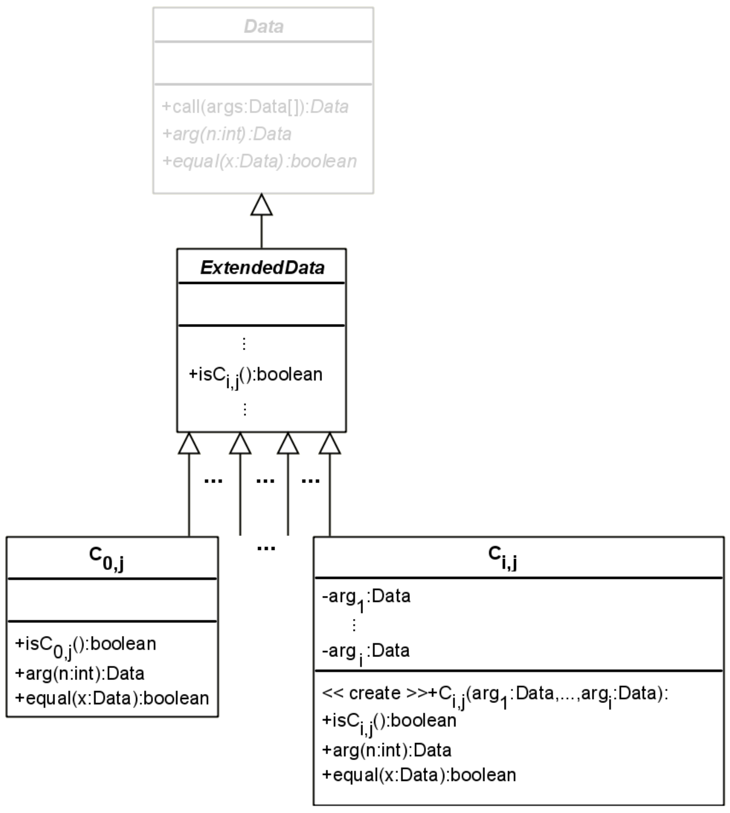

Figure 4.2: UML Diagram for the Classes of a General Theory

4.3 Time Measurements

As a first demonstration of the Theorema-Java Compiler's power, we will now show the tremendous speed-up it can achieve. Using again the representation of natural numbers given in Section 4.1, we want now to compute values of the following mathematical function **Binom4**:

$$\mathbf{Binom4}[n, m] = \left(\binom{n}{m} \bmod 4\right)$$

A recursive definition of this function, using the well known identity $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$ for $n, m > 0$, is:

$$\mathbf{Binom4}[n, m] = \begin{cases} 1 & \Leftarrow \; m = 0 \\ 1 & \Leftarrow \; n = m \\ (\mathbf{Binom4}[n-1, m-1] + \mathbf{Binom4}[n-1, m]) \bmod 4 & \Leftarrow \; \text{otherwise} \end{cases}$$

A possible Theorema implementation of this function working on the above mentioned representation of natural numbers is:

```
Definition["Mod4", any[x],
 Mod4[x++++] = Mod4[x]
 Mod4[x] = x                ]


Definition["Binom4", any[n, m],
 Binom4[n, Z] = Z+
 Binom4[n, n] = Z+
 Binom4[n+, m+] =
  Mod4[Binom4[n, m] + Binom4[n, m+]]]
```

The theory "Binom4" collects the definitions of **Mod4**, **Binom4**, and **Plus** (the latter one is taken from Section 4.1):

```
Theory["Binom4",
 Definition["Plus"]
 Definition["Mod4"]
 Definition["Binom4"]]
```

We may now compile this theory to Java by

```
Java-Theory2Java[Theory["Binom4"]]
```

and also load it:

```
Java-UseTheories[{"Binom4"}]
```

As a first example we want to compute **Binom4[15,7]**. Using the Mathematica built-in functions **Mod** and **Binomial**, we find that:

```
Mod[Binomial[15, 7], 4]
```

```
3
```

For computing **Binom4[15,7]** in a computational session of Theorema, we execute:

```
ComputationalSession[]
Use[⟨Theory["Binom4"]⟩]

Binom4[Z+++++++++++++++, Z+++++++] // AbsoluteTiming

EndComputationalSession[]
```

```
{0.3437500, ((Z+)+)+}
```

The same computation on the Java side is achieved by

Java-Compute[**Binom4**[$\mathbf{Z}^{+++++++++++++++}$, $\mathbf{Z}^{+++++++}$]] // **AbsoluteTiming**

{0.0156250, $((Z^{+})^{+})^{+}$}

Hence, the speed-up factor in this example is about 20. As a second example, let us compute **Binom4[19,9]**. Mathematica tells us

```
Mod[Binomial[19, 9], 4]
```

```
2
```

In a Theorema computational session we get

ComputationalSession[]
Use[⟨**Theory**["Binom4"]⟩]

Binom4[$\mathbf{Z}^{+++++++++++++++++++}$, $\mathbf{Z}^{+++++++++}$] // **AbsoluteTiming**

EndComputationalSession[]

{4.8906250, $(Z^{+})^{+}$}

Using the compiled version of the algorithms we are much faster:

Java-Compute[**Binom4**[$\mathbf{Z}^{+++++++++++++++++++}$, $\mathbf{Z}^{+++++++++}$]] // **AbsoluteTiming**

{0.0468750, $(Z^{+})^{+}$}

In this example the execution is around 100 times faster on the Java side than in a computational session of Theorema. Table 4.1 shows further time measurements with the function **Binom4**. The first column of this table states the computation task, and the second and third column state the numbers of seconds needed for the execution of the corresponding original Theorema code and the compiled one, respectively. The fourth column gives the speed-up factor of the compiled program with respect to the original Theorema program.

Task	Theorema	Compiled Theorema	Speed – up Factor
Binom4[15, 7]	0.33 *s*	0.02 *s*	17
Binom4[17, 8]	1.27 *s*	0.02 *s*	64
Binom4[19, 9]	4.8 *s*	0.06 *s*	80
Binom4[21, 10]	18.39 *s*	0.19 *s*	97
Binom4[25, 12]	269.48 *s*	2.7 *s*	100

Table 4.1: Time Measurements of **Binom4**

5 Translation of Sequence Variables

Sequence variables, which are variables for which an arbitrary finite number (including zero) of terms can be substituted, add expressiveness and elegance to the Theorema language. For example, all of the following terms match the pattern $\langle m, \overline{m}\rangle$: $\langle 3\rangle$, $\langle 3, a\rangle$, $\langle 3, a, a, b\rangle$, $\langle 5, 3, \langle 2, 3\rangle\rangle$. However, the empty tuple $\langle\rangle$ does not match $\langle m, \overline{m}\rangle$. Together with pattern matching, sequence variables turn out to be extremely useful in practice and lead to well structured and well readable programs in predicate logic. The problem of translating sequence variables from Theorema to Java is that such variables are not supported by the Java language. Therefore, we had to come up with a mechanism which imitates sequence variables on the Java side. This chapter is about this mechanism and how the actual translation works. Please note that, although sequence variables with all conceivable flexibility are fully supported by the Theorema language, only a certain type of patterns is supported by the current version of the Theorema-Java Compiler, namely only patterns with one sequence variable at the very end of the pattern. For example, the compiler is able to translate the pattern $\langle m, \overline{m}\rangle$, but it does not support the pattern $\langle \overline{m}, m\rangle$. Although, at first sight, this looks like a severe limitation of the compiler compared to the flexible support of sequence variables in Theorema, practice shows that sequence variables are mostly used in exactly those kinds of patterns that are supported by the compiler.

The original ideas for translating sequence variables to Java were mainly given by Bruno Buchberger ([Buch07a]).

5.1 An Example: `InsertOrdered`

Before describing the above mentioned mechanism and the general way sequence variables are translated, we want to show the usage of sequence variables and the corresponding Java source code in an example, namely the algorithm **`InsertOrdered`**, which inserts an element in a sorted list such that the list stays sorted. Here is the definition of the function **`InsertOrdered`** and the associated theory:

```
Definition["InsertOrdered", any[x, y, ȳ],
     InsertOrdered[x, ⟨⟩] = ⟨x⟩
                                { x ⌣ InsertOrdered[y, ⟨ȳ⟩]  ⇐  x < y
     InsertOrdered[x, ⟨y, ȳ⟩] = {
                                { y ⌣ InsertOrdered[x, ⟨ȳ⟩]  ⇐  otherwise      ]


Theory["InsertOrderedTheory",
    Definition["InsertOrdered"]]
```

The first parameter of this function is the element to be inserted into the list which is given in the second parameter. The function is defined by two rewrite rules which are distinguished by pattern matching on the structure of the second parameter: If the list (the second parameter) is empty, the singleton tuple with the first parameter is returned. If the list has at least one element, two cases are distinguished depending on the relative order of the first parameter and the first element of the list.

Let us look at two computations in Theorema (we have to add the packages for tuples and, because of the case distinction, for connectives to our knowledge base) :

```
Compute[InsertOrdered[13, ⟨⟩], using → ⟨Built-in["Tuples"],
   Built-in["Connectives"], Theory["InsertOrderedTheory"]⟩]
```

```
⟨13⟩
```

```
Compute[InsertOrdered[13, ⟨1, 6, 9, 14, 20, 99⟩],
 using → ⟨Built-in["Tuples"],
   Built-in["Connectives"], Theory["InsertOrderedTheory"]⟩]
```

```
⟨1, 6, 9, 13, 14, 20, 99⟩
```

To compile this Theorema theory to Java, we have to enter

```
Java-Theory2Java[Theory["InsertOrderedTheory"]]
```

The Java code of the function `InsertOrdered` looks like this:

```
public static Data insertOrdered(Data _param1,Data _param2)
{
    if ((BI_Tuple.IsTuple(_param2)&&(((Tuple)_param2).size()==0)))
    {
        return new Tuple(new Data[]{_param1});
    }//if

    if ((BI_Tuple.IsTuple(_param2)&&(((Tuple)_param2).size()≥1)))
    {
        if (Rationals.less(_param1,_param2.arg(1)))
        {
            return BI_Tuple.prepend(_param1,(Tuple)insertOrdered(_param2.arg(1),
            BI_Tuple.createTuple(new Data[]{
                BI_Tuple.restAsSequence((Tuple)_param2)})));
        }//if

        return BI_Tuple.prepend(_param2.arg(1),(Tuple)insertOrdered(_param1,
            BI_Tuple.createTuple(new Data[]{
                BI_Tuple.restAsSequence((Tuple)_param2)})));
    }//if

    return null;
}//insertOrdered
```

Please note the following features of this code:

- The first `if`-clause matches, if the second parameter is an empty tuple. This clause corresponds precisely to the first rewrite rule of the Theorema definition of the function:

```
InsertOrdered[x, ⟨⟩] = ⟨x⟩
```

- The second `if`-clause matches, if the second parameter is a tuple (`BI_Tuple.IsTuple(_param2)`) with at least one element (`((Tuple)_param2).size()≥1`). This is exactly what the left hand side of the second rewrite rule of the Theorema definition of the function tells us:

```
InsertOrdered[x, ⟨y, ȳ⟩] = ⎧x ⌣ InsertOrdered[y, ⟨ȳ⟩]  ⇐  x < y
                           ⎩y ⌣ InsertOrdered[x, ⟨ȳ⟩]  ⇐  otherwise
```

- The inner `if`-clause uses the function `less` of the class `Rationals`. The Theorema predicate **<** is automatically translated in this way because the default domain for **<** is the domain of rational numbers, which is implemented in the Java class `Rationals` (see Section 9.1).

- The class `BI_Tuple` provides the functions `restAsSequence`, which extracts all but the last elements of a tuple, and `createTuple`, which forms a tuple out of an array of `Data` elements. Used together in the way shown in the example code above, they provide a way of expressing sequence variables in Java.

We can now load the theory "InsertOrderedTheory":

```
Java-UseTheories[{"InsertOrderedTheory"}]
```

and then use the compiled code for computations:

```
Java-Compute[InsertOrdered[13, ⟨⟩]]
```

```
⟨13⟩
```

```
Java-Compute[InsertOrdered[13, ⟨1, 6, 9, 14, 20, 99⟩]]
```

```
⟨1, 6, 9, 13, 14, 20, 99⟩
```

5.2 General Translation

The translation of sequence variables into an equivalent mechanism in Java needs handling in all three stages of the general flow of translation (see Chapter 3). The intermediate language (see also Chapter 3), originally design by Tudor Jebelean, had to be extended by the author to also cope with sequence variables. Additionally, in the second step of the flow special rules have to be applied to handle these extensions correctly. And furthermore, the Java-sided framework of the Theorema-Java Compiler provides auxiliary methods to deal with these sequence-imitating structures.

5.2.1 Modifications of the Intermediate Language

The procedure of pattern matching elimination and the associated intermediate language (see Section 3.1) did originally not support sequence variables. The author added this functionality such that sequence variables are now conveniently translated into the intermediate language and, therefore, ready for being processed further.

A typical definition of a Theorema function (or predicate) which uses a sequence variable looks like this: $f[\langle x_1, \ldots, x_n, \bar{s}\rangle] = e$. (Please note again that this is the only way sequence variables are supported by the current version of the Theorema-Java Compiler, namely a single sequence variable at the very end of the pattern; see the beginning of this chapter.) x_i (for $1 \leq i \leq n$) is either a constant symbol, either a normal (i.e., non sequence) variable, or a nested expression of the form $D[..]$ for some constructor **D**. The expression e is the body of f and is defined in terms of x_i (for $1 \leq i \leq n$) and $\bar{s}$. The function **f** coded in the intermediate language looks like this:

```
•DeFun[•sig["f"], 〈_param1〉,
 •Conditional[〈〈(•const[™Tuple] = •Head[_param1]) ⋀
     (•TupleSize[_param1] ≥ n) ⋀ T, e*〉〉]]
```

The first condition in the conjunction expresses that the parameter of f has to be a tuple. The second one says that the length of this tuple has to be at least n (because the sequence variable $\bar{s}$ in $f[\langle x_1, \ldots, x_n, \bar{s}\rangle]$ matches sequences of any length including zero). T is obtained by recursively applying the methods of Section 3.1 and this section to all x_i (for $1 \leq i \leq n$) and forming the conjunction. e^* results from replacing $\bar{s}$ in e by `•seq-rest[_param1, n]`, expressing that $\bar{s}$ is obtained from *_param1* by dropping the first n elements.

If the function definition has the simple shape $f[\langle \bar{s}\rangle] = e$, the translation of **f** is even easier:

```
•DeFun[•sig["f"], 〈_param1〉,
 •Conditional[〈〈(•const[™Tuple] = •Head[_param1]), e*〉〉]]
```

e^* results from replacing $\bar{s}$ in e by `•seq[_param1]`.

5.2.2 Modifications of the Compiler

In the previous chapter we introduced two new constructs of the intermediate language: **•seq** and **•seq-rest**. We now present the necessary adaptions of the Theorema-Java Compiler in order to handle these statements and produce the corresponding Java source code. **•seq[e]**, for some expression **e**, is translated into

```
((Tuple)e*).asSequence()
```

where e^* is the translation of e and `asSequence()` is a method of the Java class `Tuple`, see also the next section. `•seq-rest[_param1, n]`, for some expression e and some integer n, is translated into

```
BI_Tuple.restAsSequence((Tuple)e*,n)
```

where e^* is the translation of **e** and `restAsSequence` is a method of the Java class `BI_Tuple`, see also the next section.

Additionally to these two adaption, the compiler has now to distinguish two cases: First, a tuple (given in its intermediate language representation with t_i being expressions)

```
•Expr[•const[™Tuple], ™Tuple[t1, …, tn]]
```

which is free of `•seq` and `•seq-rest` is translated into

```
new Tuple(new Data[]{t1*,…,tn*})
```

where $t_i{}^*$ is the translation of t_i (for $1 \le i \le n$). Secondly, if a tuple is not free of `•seq` and `•seq-rest`, it is translated into

```
BI_Tuple.createTuple(new Data[]{t1*,…,tn*})
```

where again $t_i{}^*$ is the translation of t_i (for $1 \le i \le n$).

5.2.3 Modifications of the Java Framework

As already indicated in the previous chapter, the framework of the Theorema-Java Compiler provides three methods for supporting sequence variables. The first one is `asSequence()` of the class `Tuple`:

```
public Sequence asSequence()
{
    return new Sequence(jls);
}//asSequence
```

`jls` is a global `Data` array which holds all the entries of the tuple. So, this method simply encapsulates the data of the tuple in a `Sequence` object. The second method is `restAsSequence` defined in the class `BI_Tuple` as

```
static public Sequence restAsSequence(Tuple t,int n)
{
    Data[] s = new Data[t.size()-n];
    for (int i=n;i<t.size();i++) s[i-n]=t.arg(i+1);
        return new Sequence(s);
}//restAsSequence
```

It encapsulates the entries of `t` starting with the n+1st entry in a `Sequence` object. Finally, the method `createTuple`, which is also defined in the class `BI_Tuple`, does the real work:

```
static public Tuple createTuple(Data[] jls)
{
    ArrayList<Data> al = new ArrayList<Data>();
    Data[] ts = new Data[0];

    for(int i=0;i<jls.length;i++)
    {
        if (jls[i] instanceof Sequence)
        {
            for(int j=0;j<((Sequence)jls[i]).size();j++)
                al.add(((Sequence)jls[i]).arg(j+1));
        }//if
        else
            al.add(jls[i]);
    }//for

    ts=al.toArray(ts);
    return new Tuple(ts);
}//createTuple
```

This method takes an array of `Data` objects and creates a new tuple with the entries of the array, only that entries of sequences are flattened. Let us have a look at the following example code:

```
Sequence s = new Sequence(new Data[]{BI_Integer.valueOf(18),BI_Integer.valueOf(7)});
createTuple(new Data[]{s,BI_Integer.valueOf(79)})
```

An instance `s` of the class `Sequence` is declared having the two entries `BI_Integer.valueOf(18)` and `BI_Integer.valueOf(7)`. Then, the method `BI_Tuple.createTuple` is called and a `Data` array containing `s` and `BI_Integer.valueOf(79)` is passed to it. The return value of this call is a tuple of length three having the entries `BI_Integer.valueOf(18)`, `BI_Integer.valueOf(7)`, and `BI_Integer.valueOf(79)`. So, the sequence `s` was flattened, and the element `BI_Integer.valueOf(79)` was appended.

5.3 Time Measurements

Extending the very first example of this part of the thesis (see Section 2.4), we will now implement the well known mergesort algorithm in Theorema, compile it to Java, and compare the runtime of some examples in both variants. For this, we need the following three functions:

- **`Merge`**: It takes two sorted lists of integers and merges them such that the resulting list is again sorted. For instance, **`Merge[⟨1,3,5⟩,⟨2,4⟩]`** returns **`⟨1,2,3,4,5⟩`**.
- **`SplitList`**: It takes one list, splits it into two halves, and returns a list containing these halves. For instance, **`SplitList[⟨1,2,3,4,5⟩]`** returns **`⟨⟨1,2⟩,⟨3,4,5⟩⟩`**.
- **`MergeSort`**: This function implements the mergesort algorithm. For instance,

MergeSort[⟨18,7,79,19⟩] returns **⟨7,18,19,79⟩**.

```
Definition["Merge", any[x, x̄, y, ȳ],

    Merge[⟨x̄⟩, ⟨⟩] = ⟨x̄⟩
    Merge[⟨⟩, ⟨ȳ⟩] = ⟨ȳ⟩
    Merge[⟨x, x̄⟩, ⟨y, ȳ⟩] = { x ⌣ Merge[⟨x̄⟩, ⟨y, ȳ⟩]  ⇐ x < y
                             { y ⌣ Merge[⟨x, x̄⟩, ⟨ȳ⟩]  ⇐ otherwise    ]


Definition["SplitList", any[x],
    SplitList[x] =
                   { ⟨⟨x_i | i=1,…,n/2⟩, ⟨x_i | i=n/2+1,…,n⟩⟩           ⇐ 2 ∣ n
  where[n = |x|,  {
                   { ⟨⟨x_i | i=1,…,(n-1)/2⟩, ⟨x_i | i=(n+1)/2,…,n⟩⟩     ⇐ otherwise  ]]


Definition["MergeSort", any[x],
    MergeSort[⟨x⟩] = ⟨x⟩
    MergeSort[x] = where[split = SplitList[x],
      Merge[MergeSort[split_1], MergeSort[split_2]]]]


Theory["MergeSortTheory",
      Definition["Merge"]
    Definition["SplitList"]]
    Definition["MergeSort"]
```

For the computations below we use the following knowledge base:

```
Use[⟨Built-in["Tuples"], Built-in["Numbers"],
  Built-in["Quantifiers"], Built-in["Connectives"]⟩]
```

We can now check the examples from above:

```
Compute[Merge[⟨1, 3, 5⟩, ⟨2, 4⟩], using → ⟨Theory["MergeSortTheory"]⟩]
```

```
⟨1, 2, 3, 4, 5⟩
```

```
Compute[SplitList[⟨1, 2, 3, 4, 5⟩], using → ⟨Theory["MergeSortTheory"]⟩]
```

```
⟨⟨1, 2⟩, ⟨3, 4, 5⟩⟩
```

```
Compute[MergeSort[⟨18, 7, 79, 19⟩], using → ⟨Theory["MergeSortTheory"]⟩]
```

```
⟨7, 18, 19, 79⟩
```

We may compile the theory "MergeSortTheory" to Java by

```
Java‐Theory2Java[Theory["MergeSortTheory"]]
```

and also load it:

```
Java‐UseTheories[{"MergeSortTheory"}]
```

Let us sort a list of length 100 in a Theorema computational session:

```
ComputationalSession[]
Use[⟨Theory["MergeSortTheory"]⟩]
MergeSort[⟨183, 424, 411, 78, 313, 450, 248, 181, 347, 333, 125, 254, 28,
   111, 450, 32, 480, 43, 43, 130, 302, 376, 299, 455, 263, 478, 257, 121,
   344, 467, 280, 286, 230, 156, 154, 411, 356, 261, 433, 85, 160, 74, 281,
   130, 398, 106, 494, 205, 403, 75, 430, 403, 490, 370, 170, 211, 422, 423,
   336, 391, 374, 425, 414, 311, 241, 18, 333, 500, 15, 247, 108, 207, 466,
   57, 252, 131, 368, 228, 444, 89, 181, 191, 2, 86, 472, 117, 305, 429, 31,
   189, 176, 272, 195, 253, 418, 253, 248, 124, 412, 63⟩] // AbsoluteTiming
EndComputationalSession[]
```

```
{0.4062500, ⟨2, 15, 18, 28, 31, 32, 43, 43, 57, 63, 74, 75, 78, 85, 86, 89,
  106, 108, 111, 117, 121, 124, 125, 130, 130, 131, 154, 156, 160, 170,
  176, 181, 181, 183, 189, 191, 195, 205, 207, 211, 228, 230, 241, 247,
  248, 248, 252, 253, 253, 254, 257, 261, 263, 272, 280, 281, 286, 299,
  302, 305, 311, 313, 333, 333, 336, 344, 347, 356, 368, 370, 374, 376,
  391, 398, 403, 403, 411, 411, 412, 414, 418, 422, 423, 424, 425, 429,
  430, 433, 444, 450, 450, 455, 466, 467, 472, 478, 480, 490, 494, 500⟩}
```

The same computation on the Java side is achieved by

```
Java‐Compute[
  MergeSort[⟨183, 424, 411, 78, 313, 450, 248, 181, 347, 333, 125, 254, 28,
    111, 450, 32, 480, 43, 43, 130, 302, 376, 299, 455, 263, 478, 257, 121,
    344, 467, 280, 286, 230, 156, 154, 411, 356, 261, 433, 85, 160, 74, 281,
    130, 398, 106, 494, 205, 403, 75, 430, 403, 490, 370, 170, 211, 422, 423,
    336, 391, 374, 425, 414, 311, 241, 18, 333, 500, 15, 247, 108, 207, 466,
    57, 252, 131, 368, 228, 444, 89, 181, 191, 2, 86, 472, 117, 305, 429, 31,
    189, 176, 272, 195, 253, 418, 253, 248, 124, 412, 63⟩]] // AbsoluteTiming
```

```
{0.0156250, ⟨2, 15, 18, 28, 31, 32, 43, 43, 57, 63, 74, 75, 78, 85, 86, 89,
  106, 108, 111, 117, 121, 124, 125, 130, 130, 131, 154, 156, 160, 170,
  176, 181, 181, 183, 189, 191, 195, 205, 207, 211, 228, 230, 241, 247,
  248, 248, 252, 253, 253, 254, 257, 261, 263, 272, 280, 281, 286, 299,
  302, 305, 311, 313, 333, 333, 336, 344, 347, 356, 368, 370, 374, 376,
  391, 398, 403, 403, 411, 411, 412, 414, 418, 422, 423, 424, 425, 429,
  430, 433, 444, 450, 450, 455, 466, 467, 472, 478, 480, 490, 494, 500⟩}
```

In this quite small example the execution is around 25 times faster on the Java side than in a computational session of Theorema. Table 5.1 shows further time measurements with the function **`MergeSort`**.

Task	Theorema	Compiled Theorema	Speed – up Factor
`MergeSort`[*100 elements*]	0.41 *s*	0.02 *s*	21
`MergeSort`[*200 elements*]	1.48 *s*	0.02 *s*	74
`MergeSort`[*300 elements*]	3.3 *s*	0.03 *s*	110
`MergeSort`[*500 elements*]	10.7 *s*	0.06 *s*	178
`MergeSort`[*1000 elements*]	62.53 *s*	0.19 *s*	329

Table 5.1: Time Measurements of **`MergeSort`**

6 Translation of Higher Order Functions

The Theorema-Java Compiler is able to translate functions which take one or more functions as input. Although Java does not directly support first-class functions, it is possible to overcome this limitation by encapsulating methods in objects and then pass these objects. For this mechanism to work, it is necessary that the encapsulating class is derived from a certain base class or that it implements a certain interface. In the current version of the Theorema-Java Compiler the former variant is implemented, namely all data that is passed to a compiled method is of type `Data`, which implements the method `call`, whose signature is `Data call(Data[])`.

6.1 An Example: DoubleMap

Before explaining the details of the translation of higher order functions in the above stated way, we want to give a simple example to explain the basic ideas of this translation step.

The function **DoubleMap** takes as arguments a function and a list, applies the function twice to each element of this list, and returns the resulting list. For testing **DoubleMap**, we additionally implement the functions **PlusOne** and **MapAddTwo**, and, finally, we pack everything together into the theory "Double-MapTheory".

```
Definition["PlusOne", any[x],
 PlusOne[x] = x + 1]

Definition["DoubleMap", any[x, x̄, f],
               DoubleMap[f, ⟨⟩] = ⟨⟩
 DoubleMap[f, ⟨x, x̄⟩] = f[f[x]] ⌣ DoubleMap[f, ⟨x̄⟩]]

Definition["MapAddTwo", any[x],
 MapAddTwo[x] = DoubleMap[PlusOne, x]]

Theory["MapAddTwoTheory",
  Definition["PlusOne"]
 Definition["DoubleMap"]
 Definition["MapAddTwo"]]
```

DoubleMap is a higher order function, which treats its first argument as a function and applies it to the elements of the list that comes as second argument. It is very similar to the **map** function which is very common in functional programming languages and also implemented in Mathematica as **Map**.

The following computation adds two times **1** to every element of the list ⟨**1, 2, 3, 4, 5**⟩ by applying the function **MapAddTwo** to this list:

```
Compute[MapAddTwo[⟨1, 2, 3, 4, 5⟩], using →
  ⟨Built-in["Tuples"], Built-in["Numbers"], Theory["MapAddTwoTheory"]⟩]
```

```
⟨3, 4, 5, 6, 7⟩
```

In order to do the same computation on the Java side, we first compile the theory

```
Java-Theory2Java[Theory["MapAddTwoTheory"]]
```

and then load it

```
Java-UseTheories[{"MapAddTwoTheory"}]
```

We can now use the compiled code for computations:

```
Java-Compute[MapAddTwo[⟨1, 2, 3, 4, 5⟩]]
```

```
⟨3, 4, 5, 6, 7⟩
```

The Java code of the function `DoubleMap` looks like this:

```
public static Data doubleMap(Data _param1,Data _param2)
{
    if ((BI_Tuple.IsTuple(_param2)&&(((Tuple)_param2).size()==0)))
    {
        return new Tuple(new Data[]{});
    }//if

    if ((BI_Tuple.IsTuple(_param2)&&(((Tuple)_param2).size()≥1)))
    {
        return BI_Tuple.prepend(_param1.call(new Data[]{_param1.call(
            new Data[]{_param2.arg(1)})}),(Tuple)doubleMap(_param1,BI_Tuple.
            createTuple(new Data[]{BI_Tuple.restAsSequence((Tuple)_param2,1)})));
    }//if

    return null;
}//doubleMap
```

So, the first parameter `_param1` is used as a function by calling its `call` method. To make this invocation work, we have to pass an appropriate object as first parameter. Let us have a look at how this is done in the method `mapAddTwo`:

```
public static Data mapAddTwo(Data _param1)
{
    return doubleMap(new PlusOneFunction(),_param1);
}//mapAddTwo
```

`mapAddTwo` calls the method `doubleMap` and passes to it an instance of the class
`PlusOneFunction` as first parameter. This class is automatically created by the compiler and is

implemented like this:

```
public class PlusOneFunction extends Function
{
    ExtendedData _param1=null;

    public PlusOneFunction()
    {
    }//PlusOneFunction

    public PlusOneFunction(ExtendedData _param1)
    {
        this._param1=_param1;
    }//PlusOneFunction

    public Data call(Data[] args)
    {
        if (args.length==1)
        {
            return Algorithms.plusOne(args[0]);
        }//if

        return Algorithms.plusOne(_param1);
    }//call
}//class PlusOneFunction
```

It basically contains the function `call`, which takes an array of `Data` and calls the method `Algorithms.plusOne` accordingly.

6.2 General Translation

The general translation of higher order functions and predicates is based on two ingredients: the creation of an encapsulating function class (like `PlusOneFunction` in the previous example), which is created by the Theorema-Java Compiler automatically for every function and every predicate of a theory, and the invocation of its `call` method.

Let us have a look at a simple example that is as general as possible and as complicated as necessary. The function **`f`** is defined as **`f[g,x]=g[x]`**, the function **`z`** is defined as **`z[x]=e`** (where **`e`** is an expression defined in terms of **`x`**), and the function **`a`** is defined as **`a[x]=f[z,x]`**. The function **`f`** coded in the intermediate language looks like this:

```
〈•DeFun[•sig["f"], 〈_param1, _param2〉,
  •Conditional[〈〈[∧], •Expr[_param1, 〈_param2〉]〉〉]]〉
```

The parameter *_param1* in •**Expr**[*_param1*, 〈*_param2*〉] is used as a function, and, hence, the corresponding Java code of **`f`** is:

```
public static Data f(Data _param1,Data _param2)
{
    return _param1.call(new Data[]{_param2});
}//f
```

So, the function `f` takes two parameters, invokes the `call` method of the first, and passes to it the second as an array with one entry. For making this mechanism work, the invoking method has to pass to `f` an instance of a subclass of the class `Function`, which is abstract and directly derived from `Data`. To see how this works, let us have a look at the function **a** in intermediate language:

```
•DeFun[•sig["a"], ⟨_param1⟩,
 •Conditional[⟨⟨[∧], •Expr[•const[f], ⟨•const[z], _param1⟩]⟩⟩]]
```

The corresponding Java code is:

```
public static Data a(Data _param1)
{
    return f(new zFunction(),_param1);
}//a
```

The function `a` calls `f` and passes a new instance of `zFunction`, which is automatically created by the Theorema-Java Compiler. Here is its Java code:

```
public class zFunction extends Function
{
    ExtendedData _param1=null;

    public zFunction()
    {
    }//zFunction

    public zFunction(ExtendedData _param1)
    {
        this._param1=_param1;
    }//zFunction

    public Data call(Data[] args)
    {
        if (args.length==1)
        {
            return Algorithms.z(args[0]);
        }//if

        return Algorithms.z(_param1);
    }//call
}//class zFunction
```

Hence, when `f` is called (by the method `a`), it invokes the `call` method of its first argument, which is the newly created instance of the class `zFunction`, and this method checks the number of arguments and eventually calls the method `Algorithms.z`.

6.3 Time Measurements

In this section we want to extend the mergesort example from Section 5.3 even further by implementing the algorithm as a higher order function. For this, the function **`MergeSort`** gets as second parameter a predicate defining the sorting ordering. This predicate is then passed through to **`Merge`**, which uses it to actually merge the two lists accordingly. Additionally, we define the predicates **`LessFunc`** and **`GreaterFunc`**, which implement the predicates < (less than) and > (greater than), respectively. The functions **`MergeSortIncreasing`** and **`MergeSortDecreasing`** finally put together the higher order function **`MergeSort`** and these ordering predicates.

```
Definition["OrderFunctions", any[x, y],
     LessFunc[x, y] ⇔ (x < y)
     GreaterFunc[x, y] ⇔ (x > y)]

Definition["Merge", any[x, x̄, y, ȳ, F],
     Merge[⟨x̄⟩, ⟨⟩, F] = ⟨x̄⟩
     Merge[⟨⟩, ⟨ȳ⟩, F] = ⟨ȳ⟩
     Merge[⟨x, x̄⟩, ⟨y, ȳ⟩, F] = { x ⌣ Merge[⟨x̄⟩, ⟨y, ȳ⟩, F]  ⇐ F[x, y]
                                 { y ⌣ Merge[⟨x, x̄⟩, ⟨ȳ⟩, F]  ⇐ otherwise     ]

Definition["MergeSort", any[x, F],
MergeSort[⟨x⟩, F] = ⟨x⟩
MergeSort[x, F] = where[split = SplitList[x],
  Merge[MergeSort[split₁, F], MergeSort[split₂, F], F]]]
MergeSortAscending[x] = MergeSort[x, LessFunc]
MergeSortDeccending[x] = MergeSort[x, GreaterFunc]


Theory["HigherOrderMergeSortTheory",
     Definition["OrderFunctions"]
          Definition["Merge"]
        Definition["SplitList"]      ]
        Definition["MergeSort"]
```

For the computations below we use the following knowledge base:

```
Use[⟨Built-in["Tuples"], Built-in["Numbers"],
  Built-in["Quantifiers"], Built-in["Connectives"]⟩]
```

If we want, for instance, to sort the list **`⟨99, 12, 5, 34, 9, 1, 18, 7⟩`** in ascending order, we compute

```
Compute[
  MergeSortAscending[〈183, 424, 411, 78, 313, 450, 248, 181, 347, 333, 125,
    254, 28, 111, 450, 32, 480, 43, 43, 130, 302, 376, 299, 455, 263, 478,
    257, 121, 344, 467, 280, 286, 230, 156, 154, 411, 356, 261, 433, 85, 160,
    74, 281, 130, 398, 106, 494, 205, 403, 75, 430, 403, 490, 370, 170, 211,
    422, 423, 336, 391, 374, 425, 414, 311, 241, 18, 333, 500, 15, 247, 108,
    207, 466, 57, 252, 131, 368, 228, 444, 89, 181, 191, 2, 86, 472, 117,
    305, 429, 31, 189, 176, 272, 195, 253, 418, 253, 248, 124, 412, 63〉],
  using → 〈Theory["HigherOrderMergeSortTheory"]〉] // AbsoluteTiming
```

```
{0.5312500, 〈2, 15, 18, 28, 31, 32, 43, 43, 57, 63, 74, 75, 78, 85, 86, 89,
  106, 108, 111, 117, 121, 124, 125, 130, 130, 131, 154, 156, 160, 170,
  176, 181, 181, 183, 189, 191, 195, 205, 207, 211, 228, 230, 241, 247,
  248, 248, 252, 253, 253, 254, 257, 261, 263, 272, 280, 281, 286, 299,
  302, 305, 311, 313, 333, 333, 336, 344, 347, 356, 368, 370, 374, 376,
  391, 398, 403, 403, 411, 411, 412, 414, 418, 422, 423, 424, 425, 429,
  430, 433, 444, 450, 450, 455, 466, 467, 472, 478, 480, 490, 494, 500〉}
```

To sort the same list in descending order, we call

```
Compute[MergeSortDeccending[
   〈183, 424, 411, 78, 313, 450, 248, 181, 347, 333, 125, 254, 28, 111, 450,
    32, 480, 43, 43, 130, 302, 376, 299, 455, 263, 478, 257, 121, 344,
    467, 280, 286, 230, 156, 154, 411, 356, 261, 433, 85, 160, 74, 281,
    130, 398, 106, 494, 205, 403, 75, 430, 403, 490, 370, 170, 211, 422,
    423, 336, 391, 374, 425, 414, 311, 241, 18, 333, 500, 15, 247, 108,
    207, 466, 57, 252, 131, 368, 228, 444, 89, 181, 191, 2, 86, 472, 117,
    305, 429, 31, 189, 176, 272, 195, 253, 418, 253, 248, 124, 412, 63〉],
  using → 〈Theory["HigherOrderMergeSortTheory"]〉] // AbsoluteTiming
```

```
{0.5312500, 〈500, 494, 490, 480, 478, 472, 467, 466, 455, 450, 450, 444, 433,
  430, 429, 425, 424, 423, 422, 418, 414, 412, 411, 411, 403, 403, 398,
  391, 376, 374, 370, 368, 356, 347, 344, 336, 333, 333, 313, 311, 305,
  302, 299, 286, 281, 280, 272, 263, 261, 257, 254, 253, 253, 252, 248,
  248, 247, 241, 230, 228, 211, 207, 205, 195, 191, 189, 183, 181, 181,
  176, 170, 160, 156, 154, 131, 130, 130, 125, 124, 121, 117, 111, 108,
  106, 89, 86, 85, 78, 75, 74, 63, 57, 43, 43, 32, 31, 28, 18, 15, 2〉}
```

Let us now compile and load the theory "HigherOrderMergeSortTheory" and make the same calculations on the Java side:

```
Java`Theory2Java[Theory["HigherOrderMergeSortTheory"]]
```

```
Java`UseTheories[{"HigherOrderMergeSortTheory"}]
```

```
Java-Compute[
  MergeSortAscending[⟨183, 424, 411, 78, 313, 450, 248, 181, 347, 333,
    125, 254, 28, 111, 450, 32, 480, 43, 43, 130, 302, 376, 299, 455,
    263, 478, 257, 121, 344, 467, 280, 286, 230, 156, 154, 411, 356,
    261, 433, 85, 160, 74, 281, 130, 398, 106, 494, 205, 403, 75, 430,
    403, 490, 370, 170, 211, 422, 423, 336, 391, 374, 425, 414, 311,
    241, 18, 333, 500, 15, 247, 108, 207, 466, 57, 252, 131, 368,
    228, 444, 89, 181, 191, 2, 86, 472, 117, 305, 429, 31, 189, 176,
    272, 195, 253, 418, 253, 248, 124, 412, 63⟩]] // AbsoluteTiming
```

```
{0.0156250, ⟨2, 15, 18, 28, 31, 32, 43, 43, 57, 63, 74, 75, 78, 85, 86, 89,
  106, 108, 111, 117, 121, 124, 125, 130, 130, 131, 154, 156, 160, 170,
  176, 181, 181, 183, 189, 191, 195, 205, 207, 211, 228, 230, 241, 247,
  248, 248, 252, 253, 253, 254, 257, 261, 263, 272, 280, 281, 286, 299,
  302, 305, 311, 313, 333, 333, 336, 344, 347, 356, 368, 370, 374, 376,
  391, 398, 403, 403, 411, 411, 412, 414, 418, 422, 423, 424, 425, 429,
  430, 433, 444, 450, 450, 455, 466, 467, 472, 478, 480, 490, 494, 500⟩}
```

In this quite small example the execution is around 30 times faster on the Java side than in a computational session of Theorema. Table 6.1 shows further time measurements with the function **MergeSortAscending**.

Task	Theorema	Compiled Theorema	Speed – up Factor
MergeSortAscending[*100 elements*]	0.53 *s*	0.02 *s*	27
MergeSortAscending[*200 elements*]	1.53 *s*	0.03 *s*	51
MergeSortAscending[*300 elements*]	3.4 *s*	0.03 *s*	113
MergeSortAscending[*500 elements*]	10.7 *s*	0.06 *s*	178
MergeSortAscending[*1000 elements*]	62.4 *s*	0.19 *s*	328

Table 6.1: Time Measurements of **MergeSortAscending**

7 Translation of Functors

In this thesis, a *domain* is a carrier together with operations (functions and predicates) on this carrier, and a *functor* is a function that generates a new domain from given ones. Functors provide an elegant approach to generic programming and were introduced in the Theorema system by the work of Bruno Buchberger ([Tma00]). For example, using functors, the code for the operations in the domain of polynomials over a coefficient domain needs to be written only once, independent of the specific coefficient domain. By iteration, the application of (algorithmic) functors to domains generated by (algorithmic) functors (starting from some initial, algorithmic domains) may generate a wide spectrum of (algorithmic) domains with only very little code for the few functors involved. An introduction to functors, their power, and their usage in Theorema (including a lot of examples) is given in [Buch03] and [Buch08].

The Theorema-Java Compiler is able to translate domains which are defined by the application of functors, and, for that, provides the command **`Java-DeclareDomain`**. Additionally, the framework of the compiler offers three basic domains (see Section 10.3.2): the class `Integers` representing the domain of integers, the class `Rationals` representing the domain of rational numbers, and the class `IntegersMod5` representing the domain of integers modulo five.

In this chapter of the thesis, we present first an introductory example and then explain the general translation of Theorema definitions based on functors and domains. In Chapter 11, we will present a whole case study on Gröbner Bases, where we make extensive use of functors and domains.

7.1 An Example: `CartesianProduct`

The functor **`CartesianProduct`** takes a domain **`D`** and generates the domain **`D×D`**, the cartesian product of **`D`** and **`D`**. Further details on this functor and a detailed description of the syntax and semantic can be found in [Buch03].

```
Definition["Cartesian Product", any[D],
 CartesianProduct[D] = Functor[N, any[X, x1, x2, y1, y2],

     s = ⟨⟩
     ─────────────────────────────────────────────────────────
     ∈_N [X] ⇔ (is-tuple[X] ⋀ (|X| = 2) ⋀ ∈_D [X_1] ⋀ ∈_D [X_2])
     0_N = ⟨0_D, 0_D⟩
     ⟨x1, x2⟩ >_N ⟨y1, y2⟩ ⇔ ((x1 >_D y1) ⋀ (x2 >_D y2))
     ⟨x1, x2⟩ +_N ⟨y1, y2⟩ = ⟨x1 +_D y1, x2 +_D y2⟩

     ]]

Definition["CP Domains",
 CP-Int = CartesianProduct[ℕ]]
```

Given a domain **`D`** with the decision predicate $\underset{D}{\in}$ (i.e., $\underset{D}{\in}$ **`[X]`** yields **`true`**, if and only if **`X`** is an element of the carrier of **D**), the binary predicate $\underset{D}{>}$, and the binary function $\underset{D}{+}$, **CartesianProduct[D]** returns a domain, let us call it **N**, with two predicates, one function, and a constant:

- The unary decision predicate of **N** is defined as

$$\underset{N}{\in}[X] \Leftrightarrow \left(\text{is-tuple}[X] \bigwedge (|X| = 2) \bigwedge \underset{D}{\in}[X_1] \bigwedge \underset{D}{\in}[X_2]\right)$$

That is, an arbitrary **X** is element of the carrier of the new domain **N**, if and only if **X** is a tuple, it is of length two, and its two components belong to the carrier of **D**.

- The binary predicate $\underset{N}{>}$ is defined component-wise in terms of $\underset{D}{>}$:

$$\langle x1, x2\rangle \underset{N}{>} \langle y1, y2\rangle \Leftrightarrow \left(\left(x1 \underset{D}{>} y1\right) \bigwedge \left(x2 \underset{D}{>} y2\right)\right)$$

- The constant **0** of **N** is defined as

$$\underset{N}{0} = \left\langle \underset{D}{0}, \underset{D}{0}\right\rangle$$

- The binary operation $\underset{N}{+}$ is defined component-wise in terms of $\underset{D}{+}$:

$$\langle x1, x2\rangle \underset{N}{+} \langle y1, y2\rangle = \left\langle x1 \underset{D}{+} y1, x2 \underset{D}{+} y2\right\rangle$$

Additionally, the domain **CP–Int** is defined as **CartesianProduct[ℕ]**, i.e., **CP–Int** is the cartesian product ℕ×ℕ.

We want now to do some computations in this domain in both Theorema and Java. For this, we use the following knowledge base:

```
Use[⟨Built-in["Tuples"], Built-in["Connectives"],
  Built-in["Numbers"], Built-in["Number Domains"], Built-in["Sets"],
  Definition["Cartesian Product"], Definition["CP Domains"]⟩]
```

The following computation determines whether the tuple **⟨18,7⟩** is an element of the carrier of **CP–Int**:

$$\text{Compute}\left[\underset{\text{CP-Int}}{\in}[\langle 18, 7\rangle]\right]$$

```
True
```

It returns **True**, because both **18** and **7** are natural numbers. To add **⟨1, 2⟩** and **⟨3, 4⟩** in the domain **CP–Int**, we may execute:

```
Compute[⟨1, 2⟩ +_CP-Int ⟨3, 4⟩]
```

```
⟨4, 6⟩
```

Finally, we want to check if **⟨18, 7⟩ > ⟨7, 9⟩** holds in **CP−Int**:

```
Compute[⟨18, 7⟩ >_CP-Int ⟨2, 9⟩]
```

```
False
```

This gives, of course, **False**, because **18>2** but **7<9**.

Now, we want to compile the domain **CP−Int** to Java. For this purpose, the command **Java-DeclareDomain** is provided, and a call to it has the following syntax:

```
Java-DeclareDomain[DomainName = FunctorName[Parameters], Definition]
```

DomainName is the name of the domain on the Java side, *FunctorName* is the name of the functor which is applied to *Parameters* and returns the new domain. The current version of the Theorema-Java Compiler supports *Parameters* to be a (possibly empty) sequence of domains and integers. *Definition* is the Theorema definition that defines the functor *FunctorName*.

So, in order to create the domain **CP−Int** on the Java side, we have to execute:

```
Java-DeclareDomain[CP-Int = CartesianProduct[Integers],
 Definition["Cartesian Product"]]
```

As a result, the Java class `CP_Int` (see renaming of identifiers in Section 3.2) is created and compiled:

```
public class CP_Int implements Domain
{
    public static boolean element(Data _param1)
    {
        return ((BI_Tuple.IsTuple(_param1)&&
            (BI_Integer.valueOf(((Container)_param1).size()).
                equal(BI_Integer.valueOf(2)))&&
            Integers.element(((Tuple)_param1).
                arg(((BI_Number)BI_Integer.valueOf(1)).asInt()))&&
            Integers.element(((Tuple)_param1).
                arg(((BI_Number)BI_Integer.valueOf(2)).asInt())));
    }//element

    public static boolean greater(Data _param1,Data _param2)
    {
        if ((BI_Tuple.IsTuple(_param1)&&(((Tuple)_param1).size()==2)&&
            BI_Tuple.IsTuple(_param2)&&(((Tuple)_param2).size()==2)))
        {
            return ((Integers.greater(_param1.arg(1),_param2.arg(1))&&
                        Integers.greater(_param1.arg(2),_param2.arg(2))));
        }//if

        return false;
    }//greater

    public static Data plus(Data _param1,Data _param2)
    {
        if ((BI_Tuple.IsTuple(_param1)&&(((Tuple)_param1).size()==2)&&
            BI_Tuple.IsTuple(_param2)&&(((Tuple)_param2).size()==2)))
        {
            return new Tuple(new Data[]{Integers.plus(_param1.arg(1),_param2.arg(1)),
                Integers.plus(_param1.arg(2),_param2.arg(2))});
        }//if

        return null;
    }//plus

    public static Data constants(String _param1)
    {
        if (_param1.equals("0"))
        {
            return new Tuple(new Data[]{Integers.constants("0"),
                Integers.constants("0")});
        }//if

        return null;
    }//constants

    ...
}//class CP_Int
```

This Java class has the following features:

- It implements the interface `Domain`, which is empty and which has to be implemented by all classes representing a domain.
- The functions of the Theorema domain are implemented as static methods since they are associated to the domain itself rather than to an instance of it. Actually, domain classes are never instantiated.
- The return value of methods which implement predicates is `boolean`, methods which implement functions return `Data` objects.
- The method `constants` gets a string and returns the corresponding constant.

7.2 General Translation

The general translation of a functor basically involves the translation techniques described in the previous chapters. However, additional particularities have to be considered:

- A functor, let us call it N for the moment, may contain three types of definitions, which are described in the following list:
  - A function definition has either the form $\underset{N}{f}[x_1, \ldots, x_n] = T$ or the form $\underset{N}{f}[x_1, \ldots, x_n] := T$.
  - A predicate definition has either the form $\underset{N}{p}[x_1, \ldots, x_n] \Leftrightarrow F$ or the form $\underset{N}{p}[x_1, \ldots, x_n] :\Leftrightarrow T$.
  - A constant definition has either the form $\underset{N}{C} = T$ or the form $\underset{N}{C} := T$.
- Every functor has to define a membership predicate $\in$.
- In the current version of the Theorema-Java Compiler parameters of functors may be a (possibly empty) sequence of domains and integers.
- The command **`Java–DeclareDomain[`*`DomainName`*` = `*`FunctorName`*`[`*`Parameters`*`], `*`Definition`*`]`** creates the domain ***`DomainName`*** on the Java side, i.e., it creates one Java class which contains one static method for every function and every predicate of the newly defined domain.

According to the original inventor and implementor of functors in Theorema, Bruno Buchberger, the fundamental idea of Theorema's functor concept is that Theorema (as well as Mathematica) supports general currying (see [Buch08]). Since currying is not possible in Java, we had to come up with a different concept in order to translate Theorema functors into Java code. We achieve this compilation by applying the following mechanism ([Buch07b]): Let us first define an exemplary functor **`F`** as

```
Definition["F", any[D],
 F[D] = Functor[N, any[X],
```

$$\frac{\mathbb{s} = \langle\rangle}{\underset{N}{\in}[X] \Leftrightarrow \text{true}}$$

$$\underset{N}{g}[X] = \underset{D}{h}[X]$$

```
    ]]
```

This functor takes a domain **D** and defines a function **g** in terms of **D**'s function **h**. Suppose we previously defined the domain **M**, we can create the domain **F[M]** on the Java side by executing

```
Java-DeclareDomain[FM = F[M], Definition["F"]]
```

For creating the corresponding Java code, the Theorema-Java Compiler translates every function call $\underset{D}{f}[\ldots]$ into the Java function call `M.f(…)`. So, the Java code of the method `g` in the domain `FM` looks like this:

```
public static Data g(Data _param1)
{
    return M.h(_param1);
}//g
```

Of course, this translation can equally be applied to all kinds of functions and predicates.

7.3 Time Measurements

An detailed case study of functors including time measurements in Theorema and on the Java side is presented in Chapter 11 of this thesis.

8 Calling Compiled Algorithms

The previous chapters explained explicitly the translation of Theorema definitions into executable Java code. After achieving this compilation, we have to come up with a way of calling the algorithms on the Java side from within Theorema. That is, we need an interface between Mathematica and Java which allows both instantiations of classes and calls to their methods. J/Link is the tool of our choice (see Section 2.1). The framework of the Theorema-Java Compiler provides the following three commands, which internally use a J/Link connection to an instance of the class `JavaComputer` (see Section 10.4), to call algorithms and to control the used theories and domains during the computation:

- **`Java–Compute`**: This is the main command to call Java algorithms, which were compiled from Theorema definitions beforehand. **`Java–Compute`** takes as its only argument a call to an algorithm, executes this algorithm with the given parameters, and returns its result. The call in the argument must have the same syntax as it would have in Theorema's **`Compute`** command.

- **`Java–UseTheories`**: This command is similar to Theorema's **`Use`** command. **`Java–UseTheories`** takes a list of theory names (i.e., a list of strings) and loads the corresponding Java classes, which, thereby, become available for the next calls to **`Java–Compute`**.

- **`Java–UseDomains`**: This command takes a list of domain names (i.e., a list of expressions) and loads the corresponding Java classes, which, thereby, become available for the next calls to **`Java–Compute`**.

9 Compiler Settings

The Theorema-Java Compiler provides currently two commands to change its behavior: **`Java-SetDefaultDomain`** and **`Java-SetCompilerParameter`**. This chapter describes these commands in detail.

9.1 `Java-SetDefaultDomain`

The Theorema-Java Compiler allows to use operations without stating explicitly in which domain they are to be performed. You may, for instance, evaluate **`18 + 7`** and assume that the operation **`+`** is performed in the domain of natural numbers. The command **`Java-SetDefaultDomain`** is used to calibrate this mechanism of automatically assigning a certain domain to an operator, and a call to it has the following syntax:

```
Java-SetDefaultDomain[Operation, Domain]
```

`Operation` gives the operation name or symbol, and ***`Domain`*** is the name of a previously compiled (or built-in) domain which hereby becomes the default domain of ***`Operation`***. For instance, if you want to set the default domain of the predicate ≤ to ℚ, you call

```
Java-SetDefaultDomain[ ≤ , Rationals]
```

```
{+ → Rationals, - → Rationals, * → Rationals, / → Rationals, ^ → Rationals,
 Quotient → Integers, Mod → Integers, < →
  Rationals, ≤ → Rationals, > → Rationals, ≥ → Rationals,
 Max → Rationals, Min → Rationals, ≤ → Rationals}
```

As a result, the Theorema-Java Compiler will translate every occurrence of the symbol ≤ into the Java code `Rationals.lessEqual(...)` (where `Rationals` is a Java class provided by the framework of the compiler).

To obtain a list of all operations and their assigned default domain, the command

```
Java-GetDefaultDomains[]
```

is provided. After the Theorema-Java Compiler is loaded, **`Java–GetDefaultDomains[]`** yields

```
{+ → Rationals, - → Rationals, * → Rationals, / → Rationals, ^ → Rationals,
 Quotient → Integers, Mod → Integers, < → Rationals, ≤ → Rationals, > →
  Rationals, ≥ → Rationals, Max → Rationals, Min → Rationals}
```

This means that **`Rationals`** (i.e., ℚ) is the default domain for the operations **`+`**, **`–`**, **`*`**, **`/`**, **`^`** and for the predicates <, ≤, >, ≥, **`Max`**, **`Min`**. **`Integers`** is the default domain for the operations **`Quotient`** and **`Mod`**.

9.2 `Java-SetCompilerParameter`

The compiler offers the command **`Java-SetCompilerParameter`** to alter its behavior, and a call to it has the following syntax:

```
Java-SetCompilerParameter[ParameterName, Value]
```

`ParameterName` may have one of the following values:

- **`PARAM–AUTODETECT–TAILREC`**. If ***`Value`*** is set to **`true`**, the compiler translates tail recursive function into iterative Java programs. The default value of this parameter is **`false`**.
- **`PARAM–COMPILE–SOURCE`**. If ***`Value`*** is set to **`false`**, the compiler generates Java source files, but does not compile them to Java byte code. The default value of this parameter is **`true`**.
- **`PARAM–COMPILE–DEBUGINFO`**. If ***`Value`*** is set to **`true`**, the compiler uses the option "-g" when calling the Java compiler. The default value of this parameter is **`false`**.
- **`PARAM–JAVACOMPUTER–DIRECTORY`**. ***`Value`*** sets the directory where the `JavaComputer` class (see Section 10.4) is stored.

To obtain the value of a parameter, the command **`Java-GetCompilerParameter`** is provided:

```
Java-GetCompilerParameter[ParameterName]
```

`ParameterName` may have the same values as given above.

10 The Framework of the Theorema-Java Compiler

10.1 The Package Structure

In the course of the development of the Theorema-Java Compiler a whole framework was created in order to support all the upcoming requirements of the compilation. This framework basically consists of two parts: the Theorema-Java Compiler itself, which is a Mathematica program, and a whole Java package, which basically provides Java classes for built-in structures (e.g., tuples and sequences) and built-in domains (e.g., rational numbers). This package and its subpackages have the following hierarchical structure:

```
JavaComputer
    System
        BasicDomains
        BuiltIn
        JavaComputer
    User
        Domains
        Theories
```

This chapter of the thesis is about this second part of the framework, these Java classes.

The framework of the Theorema-Java Compiler provides the precompiled Java package `JavaCompiler` which includes the *`JavaComputer`* class and two major subpackages *`BuiltIn`* and *`BasicDomains`*. All these parts are described in detail in the following sections of this chapter.

Also, the Java classes which are created during the compilation of user-defined Theorema programs are organized in subpackages of the overall package `JavaCompiler`: classes which correspond to a Theorema theory *TheoryName* (i.e., created by a call to **`Java–Theory2Java`**) belong to the package `JavaCompiler.User.Theories.`*`TheoryName`*`;` classes which represent a Theorema domain are stored in the package `JavaCompiler.User.Domains`.

10.2 The Package `BuiltIn`

The classes of the package `BuiltIn` are of general purpose and form the fundamental, Java-sided framework of the Theorema-Java Compiler. Among these classes are, for instance, the superclass of all method parameters (`Data`), boolean value expressing classes (`BooleanData`, `True`, `False`), and the `Tuple` class.

10.2.1 The Class `Data`

This abstract class is the base class of most of the classes of the framework and, in particular, of all user-defined abstract data types (see Chapter 4). Hence, it is also used as the type of all method parameters and of return values. The source code of `Data` is given below:

```
public abstract class Data
{
    public Data call(Data[] args)
    {
        return null;
    }//call
    public abstract Data arg(int n);
    public abstract boolean equal(Data x);
    public abstract String toString();
    public abstract Expr toExpr();
}//class Data
```

10.2.2 The Classes `BooleanData`, `True`, and `False`

The class `BooleanData`, an abstract subclass of `Data`, is the return value of all methods expressing a predicate and represents a boolean value. Its two subclasses `True` and `False` stand for the truth values True and False, respectively. The source codes of `BooleanData` and `True` are cited below:

```
public abstract class BooleanData extends Data
{
    public boolean isTrue()
    {
        return false;
    }//isTrue

    public boolean isFalse()
    {
        return false;
    }//isFalse
}//BooleanData
```

```
public class True extends BooleanData
{
    public boolean isTrue()
    {
        return true;
    }//isTrue

    public Data arg(int n)
    {
        return null;
    }//arg

    public boolean equal(Data x)
    {
        if (x instanceof BooleanData)
            return ((BooleanData)x).isTrue();
        else
            return false;
    }//equal

    public String toString()
    {
        return "True";
    }//toString

    public Expr toExpr()
    {
        return new Expr(Expr.SYMBOL,"True");
    }//toExpr
}//class True
```

10.2.3 The Classes `BI_Number`, `BI_Integer`, and `BI_Rational`

The class BI_Number ("BI" stands for "built-in"), which is an abstract subclass of Data, represents an element of a number domain with basic operations (e.g., addition, multiplication) and basic predicates (e.g., greater than, less than). Here is its Java source code:

```
public abstract class BI_Number extends Data
{
    public abstract BI_Number add(BI_Number a);
    public abstract BI_Number minus(BI_Number a);
    public abstract BI_Number multiply(BI_Number a);
    public abstract BI_Number divide(BI_Number a);
    public abstract BI_Number getZero();
    public abstract boolean isGreater(BI_Number a);
    public abstract boolean isGreaterEqual(BI_Number a);
    public abstract boolean isLess(BI_Number a);
    public abstract boolean isLessEqual(BI_Number a);
    public abstract boolean isUnequal(BI_Number a);
    public abstract int asInt();
}//BI_Number
```

The classes `BI_Integer` and `BI_Rational` are concrete subclasses of `BI_Number` and implement `BI_Number`'s methods accordingly. For this, `BI_Integer` internally uses an instance of the class `java.math.BigInteger`, and `BI_Rational` uses a pair of such instances.

10.2.4 The Classes `Container`, `Tuple`, and `Set`

The class `Container`, an abstract subclass of `Data`, acts as a joint superclass of `Tuple` and `Sequence`.

```
public abstract class Container extends Data
{
    public abstract int size();
}//class Container
```

An instance of the class `Tuple` corresponds to a Theorema expression whose head is **`Tuple`**, that is, it is a container for arbitrary many objects of type `Data`. Similarly, an instance of `Set` stands for a set in Theorema. Below we present the source code of `Tuple.`

```
public class Tuple extends Container
{
    Data[] jls;
    public Tuple(Data[] jls)
    {
        this.jls=jls;
    }//Tuple

    public Tuple(Data[] jls,int n)
    {
        this.jls=new Data[n];
        for(int i=0;i<n;i++)
            this.jls[i]=jls[i];
    }//Tuple

    public Data arg(int n)
    {
        if (n-1<jls.length)
            return jls[n-1];
        return null;
    }//arg

    public Data arg(BI_Integer bi)
    {
        return arg(bi.asInt());
    }//arg

    public boolean equal(Data x)
    {
        if (!(x instanceof Tuple))
            return false;

        Tuple xt=(Tuple)x;

        if (xt.size()≠jls.length)
            return false;

        for(int i=0;i<jls.length;i++)
            if (!jls[i].equal(xt.arg(i+1)))
                return false;

        return true;
    }//equal

    public Expr toExpr()
    {
        Expr[] es=new Expr[jls.length];

        for(int i=0;i<jls.length;i++)
            es[i] = jls[i].toExpr();

        return new Expr(new Expr(Expr.SYMBOL,
            String.valueOf(Constants.TRADEMARK)+"Tuple"),es);
    }//toExpr

    public int size()
    {
        return jls.length;
    }//size

    public void replace(int n,Data jl)
    {
        jls[n-1]=jl;
    }//replace

    public Sequence asSequence()
    {
        return new Sequence(jls);
    }//asSequence
    public String toString()
    {
        return toExpr().toString();
    }//toString
}//class Tuple
```

10.2.5 The Classes BI_Tuple and BI_Set

The class BI_Tuple contains several (static) methods for basic operations on tuples, e.g., appending, replacing, and deleting of elements. The source code of this class is given below.

```
public class BI_Tuple
{
    static public Data replaceElement(Tuple a1,BI_Number a2,Data a3)
    {
        int size = ((Tuple)a1).size();
        int n = a2.asInt();
        Data[] mos = new Data[size];

        for(int i=0;i<size;i++)
            if (i==n-1)
                mos[i]=a3;
            else
                mos[i]=((Tuple)a1).arg(i+1);

        return new Tuple(mos);
    }//replaceElement

    static public boolean IsTuple(Data a1)
    {
        return (a1 instanceof Tuple);
    }//IsTuple

    static public Tuple append(Tuple t,Data a)
    {
        Data[] result=new Data[t.size()+1];

        for(int i=0;i<t.size();i++)
            result[i]=t.arg(i+1);

        result[t.size()]=a;
        return new Tuple(result);
    }//append

    static public Tuple prepend(Data a,Tuple t)
    {
        Data[] result = new Data[t.size()+1];
        result[0] = a;

        for(int i=0;i<t.size();i++)
            result[i+1] = t.arg(i+1);
        return new Tuple(result);
    }//prepend

    static public Tuple append(Tuple t,Tuple ta)
    {
        int t_size = t.size();
        int ta_size = ta.size();
        Data[] result = new Data[t_size+ta_size];

        for(int i=0;i<t_size;i++)
            result[i]=t.arg(i+1);

        for(int i=0;i<ta_size;i++)
            result[t_size+i]=ta.arg(i+1);

        return new Tuple(result);
    }//append

    static public Tuple deleteElement(Tuple t,BI_Number pos)
    {
        Data[] result = new Data[t.size()-1];
        int pv = pos.asInt();

        for(int i=0;i<pv-1;i++)
            result[i]=t.arg(i+1);

        for(int i=pv;i<t.size();i++)
            result[i-1]=t.arg(i+1);

        return new Tuple(result);
    }//deleteElement

    static public Tuple deleteFirst(Tuple t)
    {
        Data[] result = new Data[t.size()-1];

        for(int i=1;i<t.size();i++)
            result[i-1]=t.arg(i+1);

        return new Tuple(result);
    }//deleteFirst

    static public Tuple insertElement(Tuple t,BI_Number pos,Data a)
    {
        Data[] result = new Data[t.size()+1];
        int pv = pos.asInt();

        for(int i=0;i<pv-1;i++)
            result[i] = t.arg(i+1);

        result[pv-1] = a;

        for(int i=pv-1;i<t.size();i++)
            result[i+1] = t.arg(i+1);
        return new Tuple(result);
    }//insertElement

    static public Tuple createTuple(Data[] jls)
    {
        ArrayList<Data>al = new ArrayList<Data>();
        Data[] ts = new Data[0];

        for(int i=0;i<jls.length;i++)
        {
            if (jls[i] instanceof Sequence)
            {
                for(int j=0;j<((Sequence)jls[i]).size();j++)
                    al.add(((Sequence)jls[i]).arg(j+1));
            }//if
            else
                al.add(jls[i]);
        }//for

        ts = al.toArray(ts);

        return new Tuple(ts);
    }//createTuple

    static public Sequence restAsSequence(Tuple t,int n)
    {
        Data[] s = new Data[t.size()-n];

        for (int i=n;i<t.size();i++)
            s[i-n] = t.arg(i+1);

        return new Sequence(s);
    }//restAsSequence
}//class BI_Tuple
```

The class `BI_Set` (its source code is given below), contains several (static) methods for basic operations on sets, e.g., intersection, union, insert.

```
public class BI_Set
{
    static public boolean IsSet(Data a1)
    {
        return (a1 instanceof Set);
    }//IsSet

    static public Data intersection(Set a1,Set a2)
    {
        ArrayList<Data>al = new ArrayList<Data>();
        Data[] ts = new Data[0];
        Data x;

        for(int i=1;i≤a1.size();i++)
        {
            x = a1.arg(i);
            if (a2.contains(x)) al.add(x);
        }//for

        ts = al.toArray(ts);

        return new Set(ts);
    }//intersection

    static public Data intersection(Data... as)
    {
        ArrayList<Data>al = new ArrayList<Data>();
        Data[] ts = new Data[0];
        Set a1 = (Set)as[0];
        int setCount = as.length;
        Data x;
        boolean b;
        int j;

        for(int i=1;i≤a1.size();i++)
        {
            x = a1.arg(i);
            j = 1;
            b = true;
            while((j<setCount)&&b)
            {
                if (!((Set)as[j]).contains(x))
                    b = false;
                else
                    j++;
            }//while

            if (b)
                al.add(x);
        }//for

        ts = al.toArray(ts);

        return new Set(ts);
    }//intersection

    static public Data union(Set a1,Set a2)
    {
        ArrayList<Data>al = new ArrayList<Data>();
        Data[] ts = new Data[0];
        Data x;

        for(int i=1;i≤a1.size();i++)
            al.add(a1.arg(i));

        for(int i=1;i≤a2.size();i++)
        {
            x = a2.arg(i);
```

```
                if (!al.contains(x))
                    al.add(x);
            }//for

            ts = al.toArray(ts);

            return new Set(ts);
        }//union

        static public Data union(Data... as)
        {
            ArrayList<Data>al = new ArrayList<Data>();
            Data[] ts = new Data[0];
            Data x;

            for(int i=0;i<as.length;i++)
                for(int j=1;j≤((Set)as[i]).size();j++)
                {
                    x = ((Set)as[i]).arg(j);
                    if (!al.contains(x))
                        al.add(x);
                }//for

            ts = al.toArray(ts);

            return new Set(ts);
        }//union

        static public Data cross(Set a1,Set a2)
        {
            int a1_size = a1.size();
            int a2_size = a2.size();
            int index = 0;
            Data[] ts = new Data[a1_size*a2_size];

            for(int i=0;i<a1_size;i++)
                for(int j=0;j<a2_size;j++)
                {
                    ts[index]=new Tuple(new Data[]{a1.arg(i+1),a2.arg(j+1)});
                    index++;
                }//for

            return new Set(ts);
        }//cross

        static public Data cross(Data... as)
        {
            int size = as.length;
            int i[] = new int[size];
            int total_size = 1;
            int p,index;
            Data[] t;
            Data[] ts;
            for(int j=0;j<size;j++)
            {
                total_size *= ((Set)as[j]).size();
                i[j] = 0;
            }//for

            if (total_size==0)
                return new Set(new Data[]{});

            ts = new Data[total_size];
            index = 0;

            while(i[0]<((Set)as[0]).size())
            {
                t = new Data[size];
                for(int j=0;j<size;j++)
                    t[j] = ((Set)as[j]).arg(i[j]+1);
                ts[index] = new Tuple(t);
                index++;
                p = size-1;
                i[p]++;
                while((p>0)&&(i[p]==((Set)as[p]).size()))
                {
                    i[p] = 0;
                    p--;
                    i[p]++;
                }//while
            }//while

            return new Set(ts);
        }//cross

        static public Data minus(Set a1,Set a2)
        {
            ArrayList<Data>al = new ArrayList<Data>();
            Data[] ts = new Data[0];
            Data x;

            for(int i=1;i≤a1.size();i++)
            {
                x = a1.arg(i);
                if (!a2.contains(x))
                    al.add(x);
            }//for

            ts = al.toArray(ts);
            return new Set(ts);
        }//minus

        static public Data minus(Data... as)
        {
            ArrayList<Data>al = new ArrayList<Data>();
            Data[] ts = new Data[0];
            Set a1 = (Set)as[0];
            int setCount = as.length;
            Data x;
            boolean b;
            int j;

            for(int i=1;i≤a1.size();i++)
            {
                x = a1.arg(i);
                j = 1;
                b = true;

                while((j<setCount)&&b)
                {
                    if (((Set)as[j]).contains(x))
                        b = false;
                    else
                        j++;
                }//while

                if (b)
                    al.add(x);
            }//for

            ts = al.toArray(ts);
            return new Set(ts);
        }//minus

        static public Set insert(Set s,Data a)
        {
            Data[] result = new Data[s.size()+1];

            for(int i=0;i<s.size();i++)
                result[i]=s.arg(i+1);

            result[s.size()]=a;
            return new Set(result);
        }//insert

        static public Data powerset(Set a)
        {
            int size = a.size();
            int p;
            int index;
            Data[] ts;

            if (size>30)//too big!
                return null;

            ts = new Data[(1<<size)];//size of ts is 2^size
            ts[0] = new Set(new Data[]{});
            p = 1;
            index = 1;

            for(int i=0;i<size;i++)
            {
                for(int j=0;j<p;j++)
                {
                    ts[index] = insert((Set)ts[j],a.arg(i+1));
                    index++;
                }//for

                p*=2;
            }//for

            return new Set(ts);
           }//powerset

        static public boolean isSubsetEqual(Set a1,Set a2)
        {
            for(int i=1;i≤a1.size();i++)
                if (!a2.contains(a1.arg(i)))
                    return false;

            return true;
           }//isSubset

          //checks if a1 is a proper(!) subset of a2
        static public boolean isSubset(Set a1,Set a2)
        {
            return ((a1.size()≠a2.size())&&isSubsetEqual(a1,a2));
           }//isSubset

        static public boolean isSupersetEqual(Set a1,Set a2)
        {
            return isSubsetEqual(a2,a1);
           }//isSupersetEqual

        static public boolean isSuperset(Set a1,Set a2)
        {
            return isSubset(a2,a1);
           }//isSuperset
       }//class BI_Set
```

10.2.6 The Class Sequence

The class `Sequence` is an auxiliary class that is used for handling an arbitrary long sequence of `Data` instances.

```
public class Sequence extends Data
{
    Data[] jls;

    public Sequence(Data... jls)
    {
        this.jls=jls;
    }//Sequence
    public Data arg(int n)
    {
        if (n-1<jls.length)
            return jls[n-1];

        return null;
    }//arg

    public Data arg(BI_Integer bi)
    {
        return arg(bi.asInt());
    }//arg

    public boolean equal(Data x)
    {
        if (!(x instanceof Sequence))
            return false;

        Sequence xt = (Sequence)x;

        if (xt.size()≠jls.length)
            return false;

        for(int i=0;i<jls.length;i++)
            if (!jls[i].equal(xt.arg(i+1)))
                return false;

        return true;
    }//equal

    public int size()
    {
        return jls.length;
    }//size

    public String toString()
    {
        return toExpr().toString();
    }//toString

    public Expr toExpr()
    {
        Expr[] es = new Expr[jls.length];

        for(int i=0;i<jls.length;i++)
            es[i] = jls[i].toExpr();

        return new Expr(new Expr(Expr.SYMBOL,"Sequence"),es);
    }//toExpr
}//class Sequence
```

10.2.7 The Class `Factory`

For every theory that is compiled by the user a class `ExtendedFactory` is created which provides information on the constructors (see Chapter 4), the functions, and the predicates of the corresponding theory. The superclass of this factory class is always `Factory`, which basically contains methods and fields for handling truth values.

```java
public class Factory
{
    static False _false=new False();
    static True _true=new True();

    public static Class[] getSignature(String methodName)
    throws NoSuchMethodException
    {
        throw new NoSuchMethodException(
            String.format("Method %s could not be found.",methodName));
    }//getSignature

    public static Data convertBooleanToData(boolean b)
    {
        if (b)
            return getTrue();
        else
            return getFalse();
    }//convertBooleanToData

    public static Data getFalse()
    {
        return _false;
    }//getFalse

    public static Data getTrue()
    {
        return _true;
    }//getTrue

    public static Data getInstance(String className,Data[] args)
    throws ClassNotFoundException
    {
        if (className.equals("False"))
            return _false;

        if (className.equals("True"))
            return _true;

        throw new ClassNotFoundException(
            String.format("Class %s with %d parameters could not be found.",
                className,args.length));
    }//getInstance
}//Factory
```

10.2.8 The Class `Constants`

This class contains several useful constants as static fields.

```
public class Constants
{
    public static final String USER_THEORIES = "JavaCompiler.User.Theories";
    public static final String USER_DOMAINS = "JavaCompiler.User.Domains";
    public static final String USER_DOMAINS_CONSTRUCTORS =
        "JavaCompiler.User.Domains._Constructors";
    public static final char TRADEMARK = 8482;
    public static final char EPSILON = 1013;
    public static final char DASH = 8211;
    public static final char INTEGERS = 63409;//dsN
    public static final char RATIONALS = 63412;//dsQ
    public static final String sDASH = String.valueOf(DASH);
}//class Constants
```

10.3 The Package `BasicDomains`

The classes of the package `BasicDomains` comprise basic and specific classes that are needed for the translation of functors and domains from Theorema to Java.

10.3.1 The Interface `Domain`

The interface `Domain` is an empty interface and has to be implemented by all classes which represent a domain.

10.3.2 The Classes `Integers`, `Rationals`, and `IntegersMod5`

The classes `Integers`, `Rationals`, and `IntegersMod5` implement the interface `Domain` and represent the domain of integers, the domain of rational numbers, and the domain of integers modulo five, respectively. Since the implementations of these classes are quite lengthy, we only present the compactly formatted code of `Rationals`:

```
public class Rationals implements Domain
{
    private static BI_Rational zero=BI_Rational.ZERO;
    private static BI_Rational one=BI_Rational.ONE;

    public static boolean isElement(Data x)
    {return ((x instanceof BI_Rational)||(x instanceof BI_Integer));}

    public static boolean element(Data x)
    {return isElement(x);}

    public static BI_Rational plus(BI_Rational a1,BI_Rational a2)
    {return a1.add(a2);}
```

```
public static BI_Rational plus(BI_Integer a1,BI_Rational a2)
{return a1.asBIRational().add(a2);}

public static BI_Rational plus(BI_Rational a1,BI_Integer a2)
{return a1.add(a2.asBIRational());}

public static BI_Rational plus(BI_Integer a1,BI_Integer a2)
{return a1.asBIRational().add(a2.asBIRational());}

public static BI_Rational plus(Data... as)
{
    BI_Rational result = BI_Rational.ZERO;
    for (int i=0;i<as.length;i++) {
        if (as[i] instanceof BI_Integer) result = plus(result,(BI_Integer)as[i]);
        else result = plus(result,(BI_Rational)as[i]);
    }//for
    return result;
}//plus

public static BI_Rational minus(BI_Integer a1,BI_Integer a2)
{return a1.minus(a2).asBIRational();}

public static BI_Rational minus(BI_Integer a1,BI_Rational a2)
{return a1.asBIRational().minus(a2);}

public static BI_Rational minus(BI_Rational a1,BI_Integer a2)
{return a1.minus(a2.asBIRational());}

public static BI_Rational minus(BI_Rational a1,BI_Rational a2)
{return a1.minus(a2);}

public static BI_Rational minus(BI_Rational a1)
{return a1.minus();}

public static BI_Rational minus(BI_Integer a1)
{return a1.minus().asBIRational();}

public static BI_Rational minus(Data... as)
{
    BI_Rational result;

    if (as[0] instanceof BI_Integer) result = ((BI_Integer)as[0]).asBIRational();
    else result = (BI_Rational)as[0];

    if (as.length==1) return result.minus();

    for (int i=1;i<as.length;i++) {
        if (as[i] instanceof BI_Integer) result = minus(result,(BI_Integer)as[i]);
        else result=minus(result,(BI_Rational)as[i]);
    }//for
    return result;
}//minus
```

```
    public static BI_Rational times(BI_Integer a1,BI_Integer a2)
    {return a1.multiply(a2).asBIRational();}

    public static BI_Rational times(BI_Integer a1,BI_Rational a2)
    {return a1.asBIRational().multiply(a2);}

    public static BI_Rational times(BI_Rational a1,BI_Integer a2)
    {return a1.multiply(a2.asBIRational());}

    public static BI_Rational times(BI_Rational a1,BI_Rational a2)
    {return a1.multiply(a2);}

    public static BI_Rational times(Data... as)
    {
        BI_Rational result=BI_Rational.ONE;
        for (int i=0;i<as.length;i++) {
            if (as[i] instanceof BI_Integer) result = times(result,(BI_Integer)as[i]);
            else result = times(result,(BI_Rational)as[i]);
        }//for
        return result;
    }//times

    public static BI_Rational divide(BI_Integer a1,BI_Integer a2)
    {return divide(a1.asBIRational(),a2.asBIRational());}

    public static BI_Rational divide(BI_Integer a1,BI_Rational a2)
    {return a1.asBIRational().divide(a2);}

    public static BI_Rational divide(BI_Rational a1,BI_Integer a2)
    {return a1.divide(a2.asBIRational());}

    public static BI_Rational divide(BI_Rational a1,BI_Rational a2)
    {return a1.divide(a2);}

    public static BI_Rational divide(Data... as)
    {
        BI_Rational result;
        if (as[0] instanceof BI_Integer) result = ((BI_Integer)as[0]).asBIRational();
        else result=(BI_Rational)as[0];

        for (int i=1;i<as.length;i++) {
            if (as[i] instanceof BI_Integer)
                result = divide(result,(BI_Integer)as[i]);
            else result = divide(result,(BI_Rational)as[i]);
        }//for
        return result;
    }//divide

    public static BI_Rational power(BI_Integer a1,BI_Integer a2)
    {return BI_Rational.valueOf(a1.pow(a2),BI_Integer.ONE);}

    public static BI_Rational power(BI_Rational a1,BI_Integer a2)
    {return a1.pow(a2);}
    public static BI_Rational power(Data... as)
    {
        BI_Rational result;
        if (as[0] instanceof BI_Rational) result = (BI_Rational)as[0];
        else result = ((BI_Integer)as[0]).asBIRational();

        for (int i=1;i<as.length;i++) {
            if (as[i] instanceof BI_Integer)
                result = power(result,(BI_Integer)as[i]);
            else if (((BI_Rational)as[i]).getDivisor().equals(BigInteger.ONE))
                result = power(result,
                    BI_Integer.valueOf(((BI_Rational)as[i]).getDividend()));
                else return null;
        }//for
        return result;
    }//power

    public static BI_Rational max(BI_Rational a1,BI_Rational a2)
    {if (a1.isGreater(a2)) return a1;else return a2;}

    public static BI_Rational max(BI_Rational a1,BI_Integer a2)
    {
        BI_Rational rat = a2.asBIRational();
        if (a1.isGreater(rat)) return a1;else return rat;
    }//max

    public static BI_Rational max(BI_Integer a1,BI_Rational a2)
    {
        BI_Rational rat = a1.asBIRational();
        if (rat.isGreater(a2)) return rat;else return a2;
    }//max

    public static BI_Rational max(BI_Integer a1,BI_Integer a2)
    {
        BI_Rational rat1 = a1.asBIRational();
        BI_Rational rat2 = a2.asBIRational();
        if (rat1.isGreater(rat2)) return rat1;else return rat2;
    }//max

    public static BI_Rational max(Data... as)
    {
        BI_Rational result;
        if (as[0] instanceof BI_Integer) result = ((BI_Integer)as[0]).asBIRational();
        else result = (BI_Rational)as[0];

        for (int i=1;i<as.length;i++) {
            if (as[i] instanceof BI_Integer) result = max(result,(BI_Integer)as[i]);
            else result = max(result,(BI_Rational)as[i]);
        }//for
        return result;
    }//max

    public static BI_Rational min(BI_Rational a1,BI_Rational a2)
    {if (a1.isGreater(a2)) return a2;else return a1;}
    public static BI_Rational min(BI_Rational a1,BI_Integer a2)
    {
        BI_Rational rat = a2.asBIRational();
        if (a1.isGreater(rat)) return rat;else return a1;
    }//min

    public static BI_Rational min(BI_Integer a1,BI_Rational a2)
    {
        BI_Rational rat = a1.asBIRational();
        if (rat.isGreater(a2)) return a2;else return rat;
    }//min

    public static BI_Rational min(BI_Integer a1,BI_Integer a2)
    {
        BI_Rational rat1 = a1.asBIRational();
        BI_Rational rat2 = a2.asBIRational();
        if (rat1.isGreater(rat2)) return rat2;else return rat1;
    }//min

    public static BI_Rational min(Data... as)
    {
        BI_Rational result;
        if (as[0] instanceof BI_Integer) result = ((BI_Integer)as[0]).asBIRational();
        else result = (BI_Rational)as[0];

        for (int i=1;i<as.length;i++) {
            if (as[i] instanceof BI_Integer) result = min(result,(BI_Integer)as[i]);
            else result = min(result,(BI_Rational)as[i]);
        }//for
        return result;
    }//min

    public static boolean less(BI_Rational a1,BI_Rational a2)
    {if (a1.isLess(a2)) return true;else return false;}

    public static boolean less(BI_Rational a1,BI_Integer a2)
    {
        BI_Rational rat = a2.asBIRational();
        if (a1.isLess(rat)) return true;else return false;
    }//less

    public static boolean less(BI_Integer a1,BI_Rational a2)
    {
        BI_Rational rat = a1.asBIRational();
        if (rat.isLess(a2)) return true;else return false;
    }//less

    public static boolean less(BI_Integer a1,BI_Integer a2)
    {
        BI_Rational rat1 = a1.asBIRational();
        BI_Rational rat2 = a2.asBIRational();
        if (rat1.isLess(rat2)) return true;else return false;
    }//less
```

```
public static boolean less(Data... as)
{
    for (int i=1;i<as.length;i++) {
        if (as[i] instanceof BI_Integer) {
            if (as[i-1] instanceof BI_Integer) {
                if (!less((BI_Integer)as[i-1],(BI_Integer)as[i])) return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!less((BI_Rational)as[i-1],(BI_Integer)as[i])) return false;
            } else return false;
        } else if (as[i] instanceof BI_Rational) {
            if (as[i-1] instanceof BI_Integer) {
                if (!less((BI_Integer)as[i-1],(BI_Rational)as[i])) return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!less((BI_Rational)as[i-1],(BI_Rational)as[i])) return false;
            } else return false;
        } else return false;
    }//for
    return true;
}//less

public static boolean greater(BI_Rational a1,BI_Rational a2)
{if (a1.isGreater(a2)) return true;else return false;}

public static boolean greater(BI_Rational a1,BI_Integer a2)
{
    BI_Rational rat = a2.asBIRational();
    if (a1.isGreater(rat)) return true;else return false;
}//greater

public static boolean greater(BI_Integer a1,BI_Rational a2)
{
    BI_Rational rat = a1.asBIRational();
    if (rat.isGreater(a2)) return true;else return false;
}//greater

public static boolean greater(BI_Integer a1,BI_Integer a2)
{
    BI_Rational rat1 = a1.asBIRational();
    BI_Rational rat2 = a2.asBIRational();
    if (rat1.isGreater(rat2)) return true;else return false;
}//greater

public static boolean greater(Data... as)
{
    for (int i=1;i<as.length;i++) {
        if (as[i] instanceof BI_Integer) {
            if (as[i-1] instanceof BI_Integer) {
                if (!greater((BI_Integer)as[i-1],(BI_Integer)as[i]))
                    return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!greater((BI_Rational)as[i-1],(BI_Integer)as[i]))
                    return false;
            } else return false;
        } else if (as[i] instanceof BI_Rational) {
            if (as[i-1] instanceof BI_Integer) {
                if (!greater((BI_Integer)as[i-1],(BI_Rational)as[i]))
                    return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!greater((BI_Rational)as[i-1],(BI_Rational)as[i]))
                    return false;
            } else return false;
        } else return false;
    }//for
    return true;
}//greater

public static boolean lessEqual(BI_Rational a1,BI_Rational a2)
{if (a1.isLessEqual(a2)) return true;else return false;}

public static boolean lessEqual(BI_Rational a1,BI_Integer a2)
{
    BI_Rational rat = a2.asBIRational();
    if (a1.isLessEqual(rat)) return true;else return false;
}//lessEqual

public static boolean lessEqual(BI_Integer a1,BI_Rational a2)
{
    BI_Rational rat = a1.asBIRational();
    if (rat.isLessEqual(a2)) return true;else return false;
}//lessEqual

public static boolean lessEqual(BI_Integer a1,BI_Integer a2)
{
    BI_Rational rat1 = a1.asBIRational();
    BI_Rational rat2 = a2.asBIRational();
    if (rat1.isLessEqual(rat2)) return true;else return false;
}//lessEqual

public static boolean lessEqual(Data... as)
{
    for (int i=1;i<as.length;i++) {
        if (as[i] instanceof BI_Integer) {
            if (as[i-1] instanceof BI_Integer) {
                if (!lessEqual((BI_Integer)as[i-1],(BI_Integer)as[i]))
                    return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!lessEqual((BI_Rational)as[i-1],(BI_Integer)as[i]))
                    return false;
            } else return false;
        } else if (as[i] instanceof BI_Rational) {
            if (as[i-1] instanceof BI_Integer) {
                if (!lessEqual((BI_Integer)as[i-1],(BI_Rational)as[i]))
                    return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!lessEqual((BI_Rational)as[i-1],(BI_Rational)as[i]))
                    return false;
            } else return false;
        } else return false;
    }//for
    return true;
}//lessEqual

public static boolean greaterEqual(BI_Rational a1,BI_Rational a2)
{if (a1.isGreaterEqual(a2)) return true;else return false;}

public static boolean greaterEqual(BI_Rational a1,BI_Integer a2)
{
    BI_Rational rat = a2.asBIRational();
    if (a1.isGreaterEqual(rat)) return true;else return false;
}//greaterEqual

public static boolean greaterEqual(BI_Integer a1,BI_Rational a2)
{
    BI_Rational rat = a1.asBIRational();
    if (rat.isGreaterEqual(a2)) return true;else return false;
}//greaterEqual

public static boolean greaterEqual(BI_Integer a1,BI_Integer a2)
{
    BI_Rational rat1 = a1.asBIRational();
    BI_Rational rat2 = a2.asBIRational();
    if (rat1.isGreaterEqual(rat2)) return true;else return false;
}//greaterEqual

public static boolean greaterEqual(Data... as)
{
    for (int i=1;i<as.length;i++) {
        if (as[i] instanceof BI_Integer) {
            if (as[i-1] instanceof BI_Integer) {
                if (!greaterEqual((BI_Integer)as[i-1],(BI_Integer)as[i]))
                    return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!greaterEqual((BI_Rational)as[i-1],(BI_Integer)as[i]))
                    return false;
            } else return false;
        } else if (as[i] instanceof BI_Rational) {
            if (as[i-1] instanceof BI_Integer) {
                if (!greaterEqual((BI_Integer)as[i-1],(BI_Rational)as[i]))
                    return false;
            } else if (as[i-1] instanceof BI_Rational) {
                if (!greaterEqual((BI_Rational)as[i-1],(BI_Rational)as[i]))
                    return false;
            } else return false;
        } else return false;
    }//for
    return true;
}//greaterEqual

public static Data constants(String name)
{
    if (name.equals("0")) return zero;
    if (name.equals("1")) return one;
    return null;
}//Constants

public static String getDomainName()
{return String.valueOf(Constants.RATIONALS);}
}//class Rationals
```

10.3.3 The Class `DomainData`

The class `DomainData` is derived from `Data` and encapsulates a domain class (e.g., the class `Rationals`), i.e., it stores a class object in its private field `domainClass`. `DomainData` is used to pass a domain as a parameter to a method, which can then use the method `call` to invoke methods of the encapsulated domain.

```
public class DomainData extends Data
{
    private Class domainClass;

    public DomainData(Class domainClass)
    {
        this.domainClass=domainClass;
    }//DomainData

    public Data call(String methodName,Class[] classes,Object[] args)
    {
        try
        {
            return (Data)((Class<?>)domainClass).
                getDeclaredMethod(methodName,classes).invoke(null,args);
        }
        catch (NoSuchMethodException ex)
        {
            Data[] ds=new Data[args.length];

            for(int i=0;i<args.length;i++)
                ds[i] = (Data)args[i];

            try
            {
                return (Data)((Class<?>)domainClass).getDeclaredMethod(methodName,
                    new Class[]{Data[].class}).invoke(null,new Object[]{ds});
            }//try
            catch (Exception exx)
            {
                exx.printStackTrace();
                return null;
            }//catch
        }//catch
        catch (Exception ex)
        {
            return null;
        }//catch
    }//call

    public Data arg(int n)
    {
        return null;
    }//arg

    public boolean equal(Data x)
    {
        return false;
    }//equal

    public String toString()
    {
        return null;
    }//toString

    public Expr toExpr()
    {
        try
        {
            return new Expr(Expr.SYMBOL,(String)((Class<?>)domainClass).
                getDeclaredMethod("getDomainName",(Class[])null).
                    invoke(null,(Object[])null));
        }//try
        catch (Exception e)
        {
            return null;
        }//catch
    }//toExpr
}//DomainData
```

10.4 The `JavaComputer` Class

The class `JavaComputer` is an interface between Theorema and Java. The three commands **`Java–Compute`**, **`Java–UseTheories`**, and **`Java–UseDomains`** (see Chapter 8) pass their arguments directly to an instance of this class, which is created during the initialization of the Theorema-Java Compiler. `JavaComputer` implements basically the following methods, which correspond to these three user commands:

- `compute`: This method takes an expression (i.e., an object of type `com.wolfram.jlink.-Expr`), evaluates it, and returns the result. It is the central method which is able to handle integers and rational numbers, to create `Tuple` objects, to instantiate encapsulating objects for higher order functions, and, of course, call methods defined in theories and domains.

- `useTheories`: This method takes a list of theory names as strings, loads the corresponding Java classes (i.e., calls the method `loadClass` of the default class loader), and builds up an internal table of the methods of the loaded theories.

`useDomains`: This method takes a list of domain names and loads (i.e., calls the method `loadClass` of the default class loader) the corresponding Java classes.

Part 2

Case Studies

In this part of the thesis we will present two case studies which show the power and usability of the Theorema-Java Compiler.

The first case study is on Gröbner Bases, and we will define there several functors to compute a so-called Gröbner Extension of a given ring. All the functors which will be presented in the course of this case study are based on the work of Bruno Buchberger in [Buch03].

In the second case study we will use Neville's algorithm for the interpolation of univariate polynomials. The functor and algorithms presented in that part are based on [Wind06].

Before we actually start, let us quit the current Mathematica kernel and reload Theorema and the Theorema-Java Compiler.

```
Needs["Theorema`"]
Needs["Theorema`JavaCompiler`JavaCompiler`"]
```

Additionally, we set Mathematica's **`$RecursionLimit`** and **`$IterationLimit`** to **`Infinity`**:

```
$RecursionLimit = Infinity;
$IterationLimit = Infinity;
```

11 Gröbner Bases

In this chapter we will present, after some preparatory work, the functor **`Groebner-Extension`** that takes a so-called reduction ring **`R`** and returns **`R`** augmented by a function which computes Gröbner Bases in **`R`**. First, we will define several auxiliary functors for computing in reduction rings and with power products represented by tuples. We will shortly explain these functors and give some exemplary computations in Theorema. Then, we will present the functor **`Groebner-Extension`**, compute Gröbner Bases in several domains, and compare the computing times of original Theorema and Java code created by the Theorema-Java Compiler.

The functors presented in this chapter were originally developed by Bruno Buchberger in [Buch03] and then adopted by the author of this thesis.

11.1 The Functor `ReductionField`

The following functor **`ReductionField`** takes a field **`D`** and adds the operations **`rdm`** and **`lcrd`**. For details on these operations we refer again to [Buch03].

$$
\begin{array}{l}
\textbf{Definition}\Big[\text{"Reduction Field"},\ \textbf{any}[\mathbf{D}],\\
\quad \textbf{ReductionField}[\mathbf{D}] = \textbf{Functor}\Big[\mathbf{N},\ \textbf{any}[x, y],\\
\qquad \underline{\mathbb{s} = \langle\rangle \qquad\qquad\qquad\qquad\qquad\qquad}\\
\\
\qquad \underset{N}{\in}[x] \Leftrightarrow \underset{D}{\in}[x]\\
\qquad \underset{N}{0} = \underset{D}{0}\\
\qquad \underset{N}{1} = \underset{D}{1}\\
\qquad x \underset{N}{+} y = x \underset{D}{+} y\\
\qquad \underset{N}{-}x = \underset{D}{-}x\\
\qquad x \underset{N}{-} y = x \underset{D}{-} y\\
\qquad x \underset{N}{*} y = x \underset{D}{*} y\\
\qquad x \underset{N}{/} y = x \underset{D}{/} y\\
\qquad x \underset{N}{>} y \Leftrightarrow x \underset{D}{>} y\\
\qquad \underset{N}{\text{rdm}}[x, y] = \begin{cases} x \underset{D}{/} y & \Leftarrow\ x \neq \underset{D}{0} \bigwedge y \neq \underset{D}{0}\\ \underset{D}{0} & \Leftarrow\ \text{otherwise}\end{cases}\\
\qquad \underset{N}{\text{lcrd}}[x, y] = \begin{cases} \underset{D}{1} & \Leftarrow\ x \neq \underset{D}{0} \bigwedge y \neq \underset{D}{0}\\ \underset{D}{0} & \Leftarrow\ \text{otherwise}\end{cases}\\
\\
\qquad \Big]\Big]
\end{array}
$$

11.2 The Functor `TuplesLex`

The functor **`TuplesLex`** takes an integer **`k`** and produces a domain of lexical ordered tuples of length **`k`**.

```
Definition["Tuples Lexical Ordering", any[k],
 TuplesLex[k] =
  Functor[N, any[x, y, x̄, ȳ],

    s = ⟨⟩
    ────────────────────────────────────────────
                    ⎧ is-tuple[x]
                    ⎪ |x| = k
    ∈ [x] ⇔ ⋀       ⎨                ⎧ ∈ [xi]
    N               ⎪       ∀        ⎪ ℕ
                    ⎩ i=1,..,k   ⋀   ⎨
                                     ⎩ xi ≥ 0
                                          ℕ

    1 = ⟨    0          |      ⟩
    N    Integers  i=1,..,k

    x * y = ⟨xi + yi     |      ⟩
      N                i=1,..,k

    x / y = ⟨xi - yi     |      ⟩
      N                i=1,..,k

    (x | y) ⟺     ∀     xi ≤ yi
       N       i=1,..,k

    lcm[x, y] = ⟨Max[xi, yi]     |      ⟩
     N                        i=1,..,k

    ⟨⟩ > ⟨⟩ ⇔ False
       N

                            ⎧ True        ⇐  x > y
    ⟨x, x̄⟩ > ⟨y, ȳ⟩ ⇔ ⎨ ⟨x̄⟩ > ⟨ȳ⟩   ⇐  (x = y)
           N                ⎪      N
                            ⎩ False       ⇐  otherwise

    deg[⟨⟩] = -1
     N

    deg[x] =    ∑     xi
     N       i=1,..,k

    isDisjunct[x, y] ⇔       ∀         ((xi = 0) ⋁ (yi = 0))
         N              i=1,..,|x|

    ]]


Theory["TuplesLex",
 Definition["Tuples Lexical Ordering"]
         TupLex2 = TuplesLex[2]
         TupLex3 = TuplesLex[3]          ]
         TupLex4 = TuplesLex[4]
```

11.3 The Functor **TuplesDeg**

The functor **TuplesDeg** takes a domain that is returned by **TuplesLex** and replaces its predicate **>** by a new one which applies the degree lexicographic ordering.

```
Definition["Tuples Degree Lexical Ordering", any[D],
 TuplesDeg[D ] =
  Functor[N, any[x, y],
```
$$\frac{\mathbb{s} = \langle\rangle}{\;}$$

$$\underset{N}{\in}[x] \Leftrightarrow \underset{D}{\in}[x]$$
$$\underset{N}{1} = \underset{D}{1}$$
$$x \underset{N}{*} y = x \underset{D}{*} y$$
$$x \underset{N}{/} y = x \underset{D}{/} y$$
$$x \underset{N}{\mid} y \Leftrightarrow x \underset{D}{\mid} y$$
$$\underset{N}{\mathrm{lcm}}[x, y] = \underset{D}{\mathrm{lcm}}[x, y]$$
$$x \underset{N}{>} y \Leftrightarrow \mathrm{where}\left[d = \underset{D}{\mathrm{deg}}[x],\ e = \underset{D}{\mathrm{deg}}[y],\ \bigvee\begin{cases} d > e \\ \bigwedge\begin{cases} d = e \\ x \underset{D}{>} y \end{cases}\end{cases}\right]$$
$$\underset{N}{\mathrm{isDisjunct}}[x, y] \Leftrightarrow \underset{D}{\mathrm{isDisjunct}}[x, y]$$

```
  ]]

Theory["TuplesDeg",
 Definition["Tuples Degree Lexical Ordering" ]
         TupDeg2 = TuplesDeg[TupLex2]
         TupDeg3 = TuplesDeg[TupLex3]            ]
         TupDeg4 = TuplesDeg[TupLex4]
```

For a better understanding of the two previous functors we want to show some exemplary computations and, for that, build the following knowledge base:

```
Use[⟨Built-in["Tuples"], Built-in["Quantifiers"],
  Built-in["Connectives"], Built-in["Numbers"],
  Built-in["Number Domains"], Theory["TuplesLex"], Theory["TuplesDeg"]⟩]
```

Multiplying two tuples means adding their entries componentwise:

$$\mathrm{Compute}\left[\langle 1, 2, 3\rangle \underset{\mathrm{TupLex3}}{*} \langle 4, 5, 6\rangle\right]$$

```
⟨5, 7, 9⟩
```

Furthermore, we my compute:

$$\mathrm{Compute}\left[\langle 1, 1, 3\rangle \underset{\mathrm{TupDeg3}}{>} \langle 1, 2, 1\rangle\right]$$

```
True
```

```
Compute[⟨1, 1, 3⟩ >_TupLex3 ⟨1, 2, 1⟩]
```

```
False
```

11.4 The Functor Poly

The functor **Poly** constructs a domain of polynomials from a given coefficient domain **C** and a given domain **T** of power products represented by tuples.

```
Definition["Polynomial Functor", any[C, T],
 Poly[C, T] =
  Functor[N, any[x, y, z, c, t, xs, x̄s̄, d, s, ys, ȳs̄, p, q, f, m̄, n̄],

    s = ⟨⟩
    ─────────────────────────────────────────────
    ∈_N[⟨⟩] ⇔ True

                                ∈_C[c]
                                ∈_T[s]
    ∈_N[⟨⟨c, s⟩, x̄s̄⟩] ⇔ ⋀ {   c ≠ 0_C
                                s >_T LPP_N[⟨x̄s̄⟩]
                                ∈_N[⟨x̄s̄⟩]

    ∈_N[xs] ⇔ False

    1_N = ⟨⟨1_C, 1_T⟩⟩
    0_N = ⟨⟩
    LPP_N[⟨⟩] = ⟨⟩
    LPP_N[⟨⟨c, s⟩, x̄s̄⟩] = s
    areLPPDisjunct_N[x, y] ⇔ isDisjunct_T[LPP_N[x], LPP_N[y]]
    lcmLPP_N[x, y] = lcm_T[LPP_N[x], LPP_N[y]]
    isTermDivisible_N[x, y] ⇔ (x |_T y)
    isTermGreater_N[x, y] ⇔ (x >_T y)
```

$$\langle\rangle \underset{N}{+} q = q$$

$$p \underset{N}{+} \langle\rangle = p$$

$$\langle\langle c, s\rangle, \bar{m}\rangle \underset{N}{+} \langle\langle d, t\rangle, \bar{n}\rangle = \begin{cases} \langle c, s\rangle \smile \left(\langle\bar{m}\rangle \underset{N}{+} \langle\langle d, t\rangle, \bar{n}\rangle\right) & \Leftarrow\ s \underset{T}{>} t \\ \langle d, t\rangle \smile \left(\langle\langle c, s\rangle, \bar{m}\rangle \underset{N}{+} \langle\bar{n}\rangle\right) & \Leftarrow\ t \underset{T}{>} s \\ \left\langle c \underset{C}{+} d, s\right\rangle \smile \left(\langle\bar{m}\rangle \underset{N}{+} \langle\bar{n}\rangle\right) & \Leftarrow\ c \neq \underset{C}{-} d \\ \langle\bar{m}\rangle \underset{N}{+} \langle\bar{n}\rangle & \Leftarrow\ \text{otherwise} \end{cases}$$

$$\underset{N}{-}\langle\rangle = \langle\rangle$$

$$\underset{N}{-}\langle\langle c, s\rangle, \bar{m}\rangle = \left\langle \underset{C}{-} c, s\right\rangle \smile \left(\underset{N}{-}\langle\bar{m}\rangle\right)$$

$$p \underset{N}{-} q = p \underset{N}{+} \left(\underset{N}{-} q\right)$$

$$p \underset{N}{*} \langle\rangle = \langle\rangle$$

$$\langle\rangle \underset{N}{*} q = \langle\rangle$$

$$\langle\langle c, s\rangle, \bar{m}\rangle \underset{N}{*} \langle\langle d, t\rangle, \bar{n}\rangle = \left(\left\langle\left\langle c \underset{C}{*} d, s \underset{T}{*} t\right\rangle\right\rangle \underset{N}{+} \langle\langle c, s\rangle\rangle \underset{N}{*} \langle\bar{n}\rangle\right) \underset{N}{+} \langle\bar{m}\rangle \underset{N}{*} \langle\langle d, t\rangle, \bar{n}\rangle$$

$$\left(\langle\rangle \underset{N}{>} p\right) \Leftrightarrow \text{False}$$

$$\left(\langle\langle c, s\rangle, \bar{m}\rangle \underset{N}{>} \langle\rangle\right) \Leftrightarrow \text{True}$$

$$\left(\langle\langle c, s\rangle, \bar{m}\rangle \underset{N}{>} \langle\langle d, t\rangle, \bar{n}\rangle\right) \Leftrightarrow \left(\bigvee \begin{cases} s \underset{T}{>} t \\ \bigwedge \begin{cases} s = t \\ c \underset{C}{>} d \end{cases} \\ \bigwedge \begin{cases} s = t \\ c = d \\ \langle\bar{m}\rangle \underset{N}{>} \langle\bar{n}\rangle \end{cases} \end{cases}\right)$$

$$\underset{N}{\text{norm}}[\langle\rangle, f] = \langle\rangle$$

$$\underset{N}{\text{norm}}[\langle\langle c, s\rangle, \bar{m}\rangle, f] = \left\langle c \underset{C}{/} f, s\right\rangle \smile \underset{N}{\text{norm}}[\langle\bar{m}\rangle, f]$$

$$\underset{N}{\text{rdm}}[\langle\rangle, p] = \underset{N}{0}$$

$$\underset{N}{\text{rdm}}[\langle\langle c, s\rangle, \bar{m}\rangle, \langle\rangle] = \underset{N}{0}$$

$$\underset{N}{\text{rdm}}[\langle\langle c, s\rangle, \bar{m}\rangle, \langle\langle d, t\rangle, \bar{n}\rangle] = \begin{cases} \left\langle\left\langle \underset{C}{\text{rdm}}[c, d], s \underset{T}{/} t\right\rangle\right\rangle & \Leftarrow\ \bigwedge \begin{cases} \underset{C}{\text{rdm}}[c, d] \neq \underset{C}{0} \\ t \underset{T}{\mid} s \end{cases} \\ \underset{N}{0} & \Leftarrow\ \text{otherwise} \end{cases}$$

$$\underset{N}{\text{lcrd}}[\langle\langle c, s\rangle, \bar{m}\rangle, \langle\langle d, t\rangle, \bar{n}\rangle] = \left\langle\left\langle \underset{C}{\text{lcrd}}[c, d], \underset{T}{\text{lcm}}[s, t]\right\rangle\right\rangle$$

]]

```
Theory["Poly",

   Definition["Reduction Field"]
  Definition["Polynomial Functor"]
       QRed = ReductionField[ℚ]
   Poly2LexQ = Poly[QRed, TupLex2]
   Poly3LexQ = Poly[QRed, TupLex3]    ]
   Poly4LexQ = Poly[QRed, TupLex4]
   Poly2DegQ = Poly[QRed, TupDeg2]
   Poly3DegQ = Poly[QRed, TupDeg3]
   Poly4DegQ = Poly[QRed, TupDeg4]
```

The theory "Poly" defines the domain **QRed** by applying the functor **ReductionField** to Theorema's built-in domain ℚ and three polynomial domains over **QRed** in two, three, and four variables. For the following computations we add this theory to our knowledge base:

```
UseAlso[⟨Theory["Poly"]⟩]
```

The two polynomials over ℚ in 3 variables $-5\,x\,y^2+2\,y\,z^2$ and $x\,z$-3 are represented by ⟨⟨−5, ⟨1, 2, 0⟩⟩, ⟨2, ⟨0, 1, 2⟩⟩⟩ and ⟨⟨1, ⟨1, 0, 1⟩⟩, ⟨−3, ⟨0, 0, 0⟩⟩⟩, respectively. Their product can be computed in Theorema:

```
Compute[
 ⟨⟨-5, ⟨1, 2, 0⟩⟩, ⟨2, ⟨0, 1, 2⟩⟩⟩ * Poly3DegQ ⟨⟨1, ⟨1, 0, 1⟩⟩, ⟨-3, ⟨0, 0, 0⟩⟩⟩]
```

```
⟨⟨-5, ⟨2, 2, 1⟩⟩, ⟨2, ⟨1, 1, 3⟩⟩, ⟨15, ⟨1, 2, 0⟩⟩, ⟨-6, ⟨0, 1, 2⟩⟩⟩
```

We may check this result with Mathematica:

```
Expand[(-5 x y^2 + 2 y z^2) * (x z - 3)]
```

$15\,x\,y^2+(-5)\,x^2\,y^2\,z+(-6)\,y\,z^2+2\,x\,y\,z^3$

We can also normalize the product, i.e., divide its coefficients by **-5**:

```
Compute[norm_Poly3DegQ[⟨⟨-5, ⟨1, 2, 0⟩⟩, ⟨2, ⟨0, 1, 2⟩⟩⟩ * Poly3DegQ
   ⟨⟨1, ⟨1, 0, 1⟩⟩, ⟨-3, ⟨0, 0, 0⟩⟩⟩, -5]]
```

$\left\langle \langle 1, \langle 2, 2, 1\rangle\rangle, \left\langle \frac{-2}{5}, \langle 1, 1, 3\rangle\right\rangle, \langle -3, \langle 1, 2, 0\rangle\rangle, \left\langle \frac{6}{5}, \langle 0, 1, 2\rangle\right\rangle\right\rangle$

11.5 The Functor `Groebner-extension`

Finally, we define the functor **`Groebner-extension`**, which takes a domain **`R`** (returned by the functor **`Poly`**) and produces a domain that provides the following three operations to compute a Gröbner Basis of a given set **`X`** of elements of **`R`**'s carrier:

- **`Gb[X]`** computes a (not necessarily reduced) Gröbner Basis of **`X`** by applying the classical, straight forward Buchberger algorithm without using Buchberger's criteria.
- **`rdGb[X]`** computes the reduced Gröbner Basis of **`X`** by computing **`Gb[X]`** and afterwards reducing the result.
- **`rdGbBC12[X]`** computes the reduced Gröbner Basis of **`X`** by applying Buchberger's algorithm and using the first and the second criterion of Buchberger.

All further details on this functor are given in [Buch03] and [Hibe95].

```
Definition["Groebner extension", any[R],
 Groebner-extension[R] =
  Functor[N, any[C, k, p, q, p̄, q̄, x, x̄, g, ḡ, X, y, ȳ, Y, h, s, c, f, M],

    𝕤 = ⟨⟩
    ─────────────────────────────────────────────

    ∈_N [X] ⇔ ∈_R [X]
    0_N = 0_R
    1_N = 1_R
    x +_N y = x +_R y
    -_N x = -_R x
    x -_N y = x -_R y
    x *_N y = x *_R y
    x >_N y ⇔ x >_R y
    rdm_N[x, y]  = rdm_R[x, y]
    lcrd_N[x, y]  = lcrd_R[x, y]
    rd_N[x, y]  = x -_N rdm_N[x, y] *_N y
     trd_N[x, Y] = trd_N[x, Y, 1]
    trd_N[x, Y, k] =
     ⎧x                                      ⇐ k > |Y|
     ⎨where[x1 = rd_N[x, Y_k],               ⇐ otherwise
     ⎪  ⎧trd_N[x1, Y, 1]    ⇐ x >_N x1
     ⎩  ⎩trd_N[x, Y, k + 1] ⇐ otherwise    ]
```

```
hrd_N[p, Y] = where[h = trd_N[p, Y], { p               ⇐ h = p
                                      { trd_N[h, Y]     ⇐ otherwise  ]
frd_N[p, Y] = frd_N[p, Y, ⟨⟩]
frd_N[p, Y, X] = frd_N[p, Y, X, hrd_N[p, Y]]
                     { X                                                   ⇐ h =_N 0
frd_N[p, Y, X, h] =  {
                     { where[lth = h_1, frd_N[h -_N ⟨lth⟩, Y, X ⌢ lth]]    ⇐ otherwise
cpd_N[x, y] = where[lxy = lcrd_N[x, y], rd_N[lxy, x] -_N rd_N[lxy, y]]
isCriterion2_N[x, y, ⟨⟩] ⇔ False
isCriterion2_N[x, y, ⟨⟨p, f, g⟩, ȳ⟩] ⇔
  { True                        ⇐  (x = f) ⋀ (y = g)
  { True                        ⇐  (x = g) ⋀ (y = f)
  { isCriterion2_N[x, y, ⟨ȳ⟩]   ⇐  otherwise
isCriterion2_N[x, y, ⟨⟩, M] ⇔ False
isCriterion2_N[x, y, ⟨g, ḡ⟩, M] ⇔
                                               x ≠ g
                                               y ≠ g
  { True                               ⇐  ⋀ {  isTermDivisible_R[LPP_R[g], lcmLPP_R[x, y]]
  {                                            Not[isCriterion2_N[x, g, M]]
                                               Not[isCriterion2_N[y, g, M]]
    isCriterion2_N[x, y, ⟨ḡ⟩, M]       ⇐  otherwise
pairs_N[⟨⟩] = ⟨⟩
pairs_N[⟨x, x̄⟩] = ⟨⟨x, ⟨x̄⟩_i⟩ | i=1,…,|⟨x̄⟩| ⟩ ≍ pairs_N[⟨x̄⟩]
ard_N[⟨⟩] = ⟨⟩
ard_N[⟨p, p̄⟩] = ard_N[⟨⟩, p, ⟨p̄⟩]
                                          { X               ⇐ h =_N 0
ard_N[X, p, ⟨⟩] = where[h = frd_N[p, X],  { X ⌢ norm_N[h]   ⇐ otherwise   ]
ard_N[X, p, ⟨q, q̄⟩] = where[h = frd_N[p, X ≍ ⟨q, q̄⟩],
   { ard_N[X, q, ⟨q̄⟩]                 ⇐  h =_N 0
   { ard_N[X ⌢ norm_N[h], q, ⟨q̄⟩]     ⇐  otherwise   ]
norm_N[⟨⟨c, s⟩, x̄⟩] = norm_R[⟨⟨c, s⟩, x̄⟩, c]
tcrd_N[x, Y] = tcrd_N[x, Y, 1, ⟨0_N | i=1,…,|Y| ⟩]
```

$$\underset{N}{\mathbf{tcrd}}[x, Y, k, C] =$$

$$\begin{cases} \langle x, C\rangle & \Leftarrow k > |Y| \\ \mathbf{where}\Big[c = \underset{N}{\mathbf{rdm}}[x, Y_k],\ x1 = x \underset{N}{-} \underset{N}{\mathbf{rdm}}[x, Y_k] \underset{N}{*} Y_k, & \Leftarrow \text{otherwise} \\ \quad \begin{cases} \underset{N}{\mathbf{tcrd}}\Big[x1, Y, 1, C_{k \leftarrow C_k \underset{N}{+} c}\Big] & \Leftarrow x \underset{N}{>} x1 \\ \underset{N}{\mathbf{tcrd}}[x, Y, k+1, C] & \Leftarrow \text{otherwise} \end{cases} \Big] & \end{cases}$$

$$\underset{N}{\mathbf{Gb}}[X] = \underset{N}{\mathbf{Gb}}\Big[X, \underset{N}{\mathbf{pairs}}[X]\Big]$$

$$\underset{N}{\mathbf{Gb}}[X, \langle\rangle] = X$$

$$\underset{N}{\mathbf{Gb}}[X, \langle\langle x, y\rangle, \bar{x}\rangle] = \mathbf{where}\Big[h = \underset{N}{\mathbf{trd}}\Big[\underset{N}{\mathbf{cpd}}[x, y], X\Big],$$

$$\begin{cases} \underset{N}{\mathbf{Gb}}[X, \langle\bar{x}\rangle] & \Leftarrow h = \underset{N}{0} \\ \underset{N}{\mathbf{Gb}}\Big[X \smallfrown h, \langle\bar{x}\rangle \asymp \Big\langle \langle X_i, h\rangle \underset{i=1,\ldots,|X|}{|} \Big\rangle\Big] & \Leftarrow \text{otherwise} \end{cases} \Big]$$

$$\underset{N}{\mathbf{GbBC12}}[X] = \underset{N}{\mathbf{GbBC12}}\Big[X, \underset{N}{\mathbf{GbBC12Aux}}\Big[\underset{N}{\mathbf{pairs}}[X], \langle\rangle\Big]\Big]$$

$$\underset{N}{\mathbf{GbBC12}}[X, \langle\rangle] = X$$

$$\underset{N}{\mathbf{GbBC12}}[X, \langle\langle p, x, y\rangle, \bar{x}\rangle] =$$

$$\begin{cases} \underset{N}{\mathbf{GbBC12Aux2}}[x, y, X, \langle\bar{x}\rangle] & \Leftarrow \bigwedge \begin{cases} \mathbf{Not}\Big[\underset{R}{\mathbf{areLPPDisjunct}}[x, y]\Big] \\ \mathbf{Not}\Big[\underset{N}{\mathbf{isCriterion2}}[x, y, X, \langle\bar{x}\rangle]\Big] \end{cases} \\ \underset{N}{\mathbf{GbBC12}}[X, \langle\bar{x}\rangle] & \Leftarrow \text{otherwise} \end{cases}$$

$$\underset{N}{\mathbf{GbBC12Aux}}[\langle\rangle, M] = M$$

$$\underset{N}{\mathbf{GbBC12Aux}}[\langle\langle x, y\rangle, \bar{x}\rangle, M] =$$

$$\underset{N}{\mathbf{GbBC12Aux}}\Big[\langle\bar{x}\rangle, \underset{N}{\mathbf{GbBC12Aux}}\Big[\underset{R}{\mathbf{lcmLPP}}[x, y], x, y, M\Big]\Big]$$

$$\underset{N}{\mathbf{GbBC12Aux}}[p, x, y, \langle\rangle] = \langle\langle p, x, y\rangle\rangle$$

$$\underset{N}{\mathbf{GbBC12Aux}}[p, x, y, \langle\langle q, f, g\rangle, \bar{x}\rangle] =$$

$$\begin{cases} \langle p, x, y\rangle \smallsmile \langle\langle q, f, g\rangle, \bar{x}\rangle & \Leftarrow \underset{R}{\mathbf{isTermGreater}}[q, p] \\ \langle q, f, g\rangle \smallsmile \underset{N}{\mathbf{GbBC12Aux}}[p, x, y, \langle\bar{x}\rangle] & \Leftarrow \text{otherwise} \end{cases}$$

$$\underset{N}{\mathbf{GbBC12Aux2}}[x, y, X, M] = \mathbf{where}\Big[h = \underset{N}{\mathbf{trd}}\Big[\underset{N}{\mathbf{cpd}}[x, y], X\Big],$$

$$\begin{cases} \underset{N}{\mathbf{GbBC12}}[X, M] & \Leftarrow h = \underset{N}{0} \\ \underset{N}{\mathbf{GbBC12}}\Big[h \smallsmile X, \underset{N}{\mathbf{GbBC12Aux2}}[h, X, M]\Big] & \Leftarrow \text{otherwise} \end{cases} \Big]$$

$$\underset{N}{\mathbf{GbBC12Aux2}}[h, \langle\rangle, M] = M$$

$$\underset{N}{\mathbf{GbBC12Aux2}}[h, \langle x, \bar{x}\rangle, M] =$$

$$\underset{N}{\mathbf{GbBC12Aux2}}\Big[h, \langle\bar{x}\rangle, \underset{N}{\mathbf{GbBC12Aux}}\Big[\underset{R}{\mathbf{lcmLPP}}[h, x], h, x, M\Big]\Big]$$

```
    rdGb[X] = ard[Gb[X]]
       N       N   N
    rdGbBC12[X] = ard[GbBC12[X]]
        N          N     N
      ]]
```

```
Theory["GB",
    Definition["Groebner extension"]
 GB2LexQ = Groebner-extension[Poly2LexQ]
 GB2DegQ = Groebner-extension[Poly2DegQ]
 GB3LexQ = Groebner-extension[Poly3LexQ] ]
 GB3DegQ = Groebner-extension[Poly3DegQ]
 GB4LexQ = Groebner-extension[Poly4LexQ]
 GB4DegQ = Groebner-extension[Poly4DegQ]
```

We add the theory "GB" to our current knowledge base:

```
UseAlso[〈Theory["GB"]〉]
```

We may now compute, for instance, the reduced Gröbner Basis of two polynomials over $\mathbb{Q}$ with respect to the lexicographic term ordering:

```
Compute[ rdGb    [〈〈〈1, 〈1, 0〉〉, 〈-1, 〈0, 1〉〉, 〈-5, 〈0, 0〉〉〉,
        GB2LexQ
   〈〈1, 〈1, 1〉〉, 〈-1, 〈1, 0〉〉, 〈3, 〈0, 0〉〉〉〉]]
```

```
〈〈〈1, 〈1, 0〉〉, 〈-1, 〈0, 1〉〉, 〈-5, 〈0, 0〉〉〉,
 〈〈1, 〈0, 2〉〉, 〈4, 〈0, 1〉〉, 〈-2, 〈0, 0〉〉〉〉
```

Hence, the reduced Gröbner Basis of $\{x - y - 5, x\,y - x + 3\}$ is $\{x - y - 5, y^2 + 4\,y - 2\}$.

11.6 Compilation to Java

Now, we want to create all these domains which we defined in Theorema also on the Java side:

```
Java-DeclareDomain[TupLex2 = TuplesLex[2],
 Definition["Tuples Lexical Ordering"]]
Java-DeclareDomain[TupLex3 = TuplesLex[3],
 Definition["Tuples Lexical Ordering"]]
Java-DeclareDomain[TupLex4 = TuplesLex[4],
 Definition["Tuples Lexical Ordering"]]
```

```
Java-DeclareDomain[TupDeg2 = TuplesDeg[TupLex2],
 Definition["Tuples Degree Lexical Ordering" ]]
Java-DeclareDomain[TupDeg3 = TuplesDeg[TupLex3],
 Definition["Tuples Degree Lexical Ordering" ]]
Java-DeclareDomain[TupDeg4 = TuplesDeg[TupLex4],
 Definition["Tuples Degree Lexical Ordering" ]]
```

```
Java-DeclareDomain[QRed = ReductionField[ℚ],
 Definition["Reduction Field"]]
```

```
Java-DeclareDomain[Poly2LexQ = Poly[QRed, TupLex2],
 Definition["Polynomial Functor"]]
Java-DeclareDomain[Poly3LexQ = Poly[QRed, TupLex3],
 Definition["Polynomial Functor"]]
Java-DeclareDomain[Poly4LexQ = Poly[QRed, TupLex4],
 Definition["Polynomial Functor"]]
Java-DeclareDomain[Poly2DegQ = Poly[QRed, TupDeg2],
 Definition["Polynomial Functor"]]
Java-DeclareDomain[Poly3DegQ = Poly[QRed, TupDeg3],
 Definition["Polynomial Functor"]]
Java-DeclareDomain[Poly4DegQ = Poly[QRed, TupDeg4],
 Definition["Polynomial Functor"]]
```

```
Java-DeclareDomain[GB2LexQ = Groebner-extension[Poly2LexQ],
 Definition["Groebner extension"]]
Java-DeclareDomain[GB3LexQ = Groebner-extension[Poly3LexQ],
 Definition["Groebner extension"]]
Java-DeclareDomain[GB3DegQ = Groebner-extension[Poly3DegQ],
 Definition["Groebner extension"]]
```

11.7 Timing Measurements

11.7.1 The First Experiment

Finally, the stage for demonstrating some time measurements is set on both the Theorema side and the Java side. We start with some computations in Theorema. Please note that, since we want to compute in a computational session of Theorema, we have to set up the knowledge base accordingly.

```
Use[〈Built-in["Tuples"],
  Built-in["Quantifiers"], Built-in["Connectives"],
  Built-in["Numbers"], Built-in["Number Domains"]〉]

ComputationalSession[];
Use[〈Theory["TuplesLex"],
  Theory["TuplesDeg"], Theory["Poly"], Theory["GB"]〉]
```

```
rdGb       [〈〈〈1, 〈1, 0〉〉, 〈-1, 〈0, 1〉〉, 〈-5, 〈0, 0〉〉〉,
GB2LexQ
   〈〈1, 〈1, 1〉〉, 〈-1, 〈1, 0〉〉, 〈3, 〈0, 0〉〉〉〉] // AbsoluteTiming
```

```
{0.2031250, 〈〈〈1, 〈1, 0〉〉, 〈-1, 〈0, 1〉〉, 〈-5, 〈0, 0〉〉〉,
  〈〈1, 〈0, 2〉〉, 〈4, 〈0, 1〉〉, 〈-2, 〈0, 0〉〉〉〉〉}
```

```
EndComputationalSession[]
```

Now, we want to do the same computation by using the compiled Java program:

```
Java-UseDomains[{GB2LexQ, GB3LexQ, GB3DegQ}]
```

```
Java-Compute[rdGb       [〈〈〈1, 〈1, 0〉〉, 〈-1, 〈0, 1〉〉, 〈-5, 〈0, 0〉〉〉,
             GB2LexQ
    〈〈1, 〈1, 1〉〉, 〈-1, 〈1, 0〉〉, 〈3, 〈0, 0〉〉〉〉]] // AbsoluteTiming
```

```
{0.0156250, 〈〈〈1, 〈1, 0〉〉, 〈-1, 〈0, 1〉〉, 〈-5, 〈0, 0〉〉〉,
  〈〈1, 〈0, 2〉〉, 〈4, 〈0, 1〉〉, 〈-2, 〈0, 0〉〉〉〉〉}
```

So, the speed-up factor in this example is about 13.

11.7.2 The Second Experiment

In this experiment we want to compute the reduced Gröbner Basis of a set of polynomials in three variables over the rational numbers:

```
ComputationalSession[]
rdGbBC12  [〈〈〈-2, 〈2, 0, 1〉〉, 〈1/3, 〈0, 1, 1〉〉, 〈5/7, 〈0, 1, 0〉〉〉,
 GB3LexQ
  〈〈1/2, 〈1, 1, 1〉〉, 〈-3, 〈1, 0, 1〉〉, 〈1, 〈0, 0, 0〉〉〉〉] // AbsoluteTiming

EndComputationalSession[]
```

```
{0.5937500,
 〈〈〈1, 〈0, 3, 2〉〉, 〈15/7, 〈0, 3, 1〉〉, 〈-12, 〈0, 2, 2〉〉, 〈-180/7, 〈0, 2, 1〉〉,
   〈36, 〈0, 1, 2〉〉, 〈540/7, 〈0, 1, 1〉〉, 〈-24, 〈0, 0, 0〉〉〉, 〈〈1, 〈1, 0, 0〉〉,
   〈1/12, 〈0, 2, 1〉〉, 〈5/28, 〈0, 2, 0〉〉, 〈-1/2, 〈0, 1, 1〉〉, 〈-15/14, 〈0, 1, 0〉〉〉〉〉}
```

```
Java-Compute[rdGbBC12    [⟨⟨⟨-2, ⟨2, 0, 1⟩⟩, ⟨1/3, ⟨0, 1, 1⟩⟩, ⟨5/7, ⟨0, 1, 0⟩⟩⟩,
                 GB3LexQ
    ⟨⟨1/2, ⟨1, 1, 1⟩⟩, ⟨-3, ⟨1, 0, 1⟩⟩, ⟨1, ⟨0, 0, 0⟩⟩⟩⟩]] // AbsoluteTiming
```

```
{0.0156250,
 ⟨⟨⟨1, ⟨0, 3, 2⟩⟩, ⟨15/7, ⟨0, 3, 1⟩⟩, ⟨-12, ⟨0, 2, 2⟩⟩, ⟨-180/7, ⟨0, 2, 1⟩⟩,
   ⟨36, ⟨0, 1, 2⟩⟩, ⟨540/7, ⟨0, 1, 1⟩⟩, ⟨-24, ⟨0, 0, 0⟩⟩⟩, ⟨⟨1, ⟨1, 0, 0⟩⟩,
   ⟨1/12, ⟨0, 2, 1⟩⟩, ⟨5/28, ⟨0, 2, 0⟩⟩, ⟨-1/2, ⟨0, 1, 1⟩⟩, ⟨-15/14, ⟨0, 1, 0⟩⟩⟩⟩⟩}
```

In this experiment the speed-up factor is about 38.

11.7.3 The Third Experiment

```
ComputationalSession[]

rdGbBC12    [⟨⟨⟨-2, ⟨2, 0, 1⟩⟩, ⟨1/3, ⟨0, 1, 1⟩⟩, ⟨5/7, ⟨0, 1, 0⟩⟩⟩,
  GB3DegQ
   ⟨⟨1/2, ⟨1, 1, 1⟩⟩, ⟨-3, ⟨1, 0, 1⟩⟩, ⟨1, ⟨0, 0, 0⟩⟩⟩, ⟨⟨-1/5, ⟨3, 1, 0⟩⟩,
    ⟨2/3, ⟨1, 2, 0⟩⟩, ⟨1, ⟨0, 1, 1⟩⟩, ⟨-3, ⟨0, 0, 0⟩⟩⟩⟩] // AbsoluteTiming

EndComputationalSession[]
```

$$\{42.3125000,$$
$$\langle\langle\langle 1, \langle 0, 0, 3\rangle\rangle, \langle \frac{257}{525}, \langle 2, 0, 0\rangle\rangle, \langle \frac{-2243}{4410}, \langle 1, 1, 0\rangle\rangle, \langle \frac{53\,642}{1575}, \langle 1, 0, 1\rangle\rangle,$$
$$\langle \frac{-6227}{3087}, \langle 0, 2, 0\rangle\rangle, \langle \frac{-361}{150}, \langle 0, 1, 1\rangle\rangle, \langle \frac{4171}{630}, \langle 0, 0, 2\rangle\rangle,$$
$$\langle \frac{-127\,088}{11\,025}, \langle 1, 0, 0\rangle\rangle, \langle \frac{331\,627}{30\,870}, \langle 0, 1, 0\rangle\rangle, \langle \frac{-1037}{210}, \langle 0, 0, 1\rangle\rangle,$$
$$\langle \frac{-41\,942}{4725}, \langle 0, 0, 0\rangle\rangle\rangle, \langle\langle 1, \langle 0, 3, 0\rangle\rangle, \langle \frac{12\,348}{11\,975}, \langle 2, 0, 0\rangle\rangle,$$
$$\langle \frac{55\,202}{11\,975}, \langle 1, 1, 0\rangle\rangle, \langle \frac{642\,096}{11\,975}, \langle 1, 0, 1\rangle\rangle, \langle \frac{-5979}{479}, \langle 0, 2, 0\rangle\rangle,$$
$$\langle \frac{37\,044}{2395}, \langle 0, 0, 2\rangle\rangle, \langle \frac{-17\,388}{479}, \langle 1, 0, 0\rangle\rangle, \langle \frac{84\,918}{2395}, \langle 0, 1, 0\rangle\rangle,$$
$$\langle \frac{-26\,754}{2395}, \langle 0, 0, 1\rangle\rangle, \langle \frac{-146\,412}{11\,975}, \langle 0, 0, 0\rangle\rangle\rangle,$$
$$\langle\langle 1, \langle 1, 0, 2\rangle\rangle, \langle \frac{-1}{10}, \langle 2, 0, 0\rangle\rangle, \langle \frac{3}{28}, \langle 1, 1, 0\rangle\rangle, \langle \frac{-549}{70}, \langle 1, 0, 1\rangle\rangle,$$
$$\langle \frac{2395}{4116}, \langle 0, 2, 0\rangle\rangle, \langle \frac{19}{30}, \langle 0, 1, 1\rangle\rangle, \langle \frac{-3}{2}, \langle 0, 0, 2\rangle\rangle, \langle \frac{2444}{735}, \langle 1, 0, 0\rangle\rangle,$$
$$\langle \frac{-3323}{1029}, \langle 0, 1, 0\rangle\rangle, \langle \frac{3}{4}, \langle 0, 0, 1\rangle\rangle, \langle \frac{19}{10}, \langle 0, 0, 0\rangle\rangle\rangle,$$
$$\langle\langle 1, \langle 3, 0, 0\rangle\rangle, \langle \frac{3}{2}, \langle 2, 0, 0\rangle\rangle, \langle \frac{-415}{84}, \langle 1, 1, 0\rangle\rangle,$$
$$\langle 78, \langle 1, 0, 1\rangle\rangle, \langle \frac{-225}{28}, \langle 0, 1, 1\rangle\rangle, \langle \frac{45}{2}, \langle 0, 0, 2\rangle\rangle,$$
$$\langle -5, \langle 0, 1, 0\rangle\rangle, \langle \frac{895}{28}, \langle 0, 0, 1\rangle\rangle, \langle \frac{-57}{2}, \langle 0, 0, 0\rangle\rangle\rangle,$$
$$\langle\langle 1, \langle 2, 1, 0\rangle\rangle, \langle -30, \langle 1, 0, 1\rangle\rangle, \langle \frac{2395}{294}, \langle 0, 2, 0\rangle\rangle, \langle \frac{-15}{2}, \langle 0, 1, 1\rangle\rangle,$$
$$\langle \frac{450}{7}, \langle 1, 0, 0\rangle\rangle, \langle \frac{-3375}{49}, \langle 0, 1, 0\rangle\rangle, \langle 45, \langle 0, 0, 1\rangle\rangle\rangle,$$
$$\langle\langle 1, \langle 0, 1, 2\rangle\rangle, \langle \frac{2}{5}, \langle 2, 0, 0\rangle\rangle, \langle \frac{-3}{7}, \langle 1, 1, 0\rangle\rangle, \langle \frac{114}{5}, \langle 1, 0, 1\rangle\rangle,$$
$$\langle \frac{-4}{3}, \langle 0, 1, 0\rangle\rangle, \langle -3, \langle 0, 0, 1\rangle\rangle, \langle \frac{-38}{5}, \langle 0, 0, 0\rangle\rangle\rangle,$$
$$\langle\langle 1, \langle 1, 2, 0\rangle\rangle, \langle \frac{28}{5}, \langle 2, 0, 0\rangle\rangle, \langle -6, \langle 1, 1, 0\rangle\rangle, \langle \frac{-14}{15}, \langle 0, 1, 0\rangle\rangle\rangle,$$
$$\langle\langle 1, \langle 0, 2, 1\rangle\rangle, \langle \frac{15}{7}, \langle 0, 2, 0\rangle\rangle, \langle -6, \langle 0, 1, 1\rangle\rangle, \langle 12, \langle 1, 0, 0\rangle\rangle,$$
$$\langle \frac{-90}{7}, \langle 0, 1, 0\rangle\rangle\rangle, \langle\langle 1, \langle 2, 0, 1\rangle\rangle, \langle \frac{-1}{6}, \langle 0, 1, 1\rangle\rangle, \langle \frac{-5}{14}, \langle 0, 1, 0\rangle\rangle\rangle,$$
$$\langle\langle 1, \langle 1, 1, 1\rangle\rangle, \langle -6, \langle 1, 0, 1\rangle\rangle, \langle 2, \langle 0, 0, 0\rangle\rangle\rangle\rangle\}$$

```
Java-Compute[rdGbBC12_GB3DegQ[⟨⟨⟨-2, ⟨2, 0, 1⟩⟩, ⟨1/3, ⟨0, 1, 1⟩⟩, ⟨5/7, ⟨0, 1, 0⟩⟩⟩,
    ⟨⟨1/2, ⟨1, 1, 1⟩⟩, ⟨-3, ⟨1, 0, 1⟩⟩, ⟨1, ⟨0, 0, 0⟩⟩⟩, ⟨⟨-1/5, ⟨3, 1, 0⟩⟩,
     ⟨2/3, ⟨1, 2, 0⟩⟩, ⟨1, ⟨0, 1, 1⟩⟩, ⟨-3, ⟨0, 0, 0⟩⟩⟩⟩]] // AbsoluteTiming
```

```
{0.4843750,
 ⟨⟨⟨1, ⟨0, 0, 3⟩⟩, ⟨257/525, ⟨2, 0, 0⟩⟩, ⟨-2243/4410, ⟨1, 1, 0⟩⟩, ⟨53 642/1575, ⟨1, 0, 1⟩⟩,
   ⟨-6227/3087, ⟨0, 2, 0⟩⟩, ⟨-361/150, ⟨0, 1, 1⟩⟩, ⟨4171/630, ⟨0, 0, 2⟩⟩,
   ⟨-127 088/11 025, ⟨1, 0, 0⟩⟩, ⟨331 627/30 870, ⟨0, 1, 0⟩⟩, ⟨-1037/210, ⟨0, 0, 1⟩⟩,
   ⟨-41 942/4725, ⟨0, 0, 0⟩⟩⟩, ⟨⟨1, ⟨0, 3, 0⟩⟩, ⟨12 348/11 975, ⟨2, 0, 0⟩⟩,
   ⟨55 202/11 975, ⟨1, 1, 0⟩⟩, ⟨642 096/11 975, ⟨1, 0, 1⟩⟩, ⟨-5979/479, ⟨0, 2, 0⟩⟩,
   ⟨37 044/2395, ⟨0, 0, 2⟩⟩, ⟨-17 388/479, ⟨1, 0, 0⟩⟩, ⟨84 918/2395, ⟨0, 1, 0⟩⟩,
   ⟨-26 754/2395, ⟨0, 0, 1⟩⟩, ⟨-146 412/11 975, ⟨0, 0, 0⟩⟩⟩,
  ⟨⟨1, ⟨1, 0, 2⟩⟩, ⟨-1/10, ⟨2, 0, 0⟩⟩, ⟨3/28, ⟨1, 1, 0⟩⟩, ⟨-549/70, ⟨1, 0, 1⟩⟩,
   ⟨2395/4116, ⟨0, 2, 0⟩⟩, ⟨19/30, ⟨0, 1, 1⟩⟩, ⟨-3/2, ⟨0, 0, 2⟩⟩, ⟨2444/735, ⟨1, 0, 0⟩⟩,
   ⟨-3323/1029, ⟨0, 1, 0⟩⟩, ⟨3/4, ⟨0, 0, 1⟩⟩, ⟨19/10, ⟨0, 0, 0⟩⟩⟩,
  ⟨⟨1, ⟨3, 0, 0⟩⟩, ⟨3/2, ⟨2, 0, 0⟩⟩, ⟨-415/84, ⟨1, 1, 0⟩⟩,
   ⟨78, ⟨1, 0, 1⟩⟩, ⟨-225/28, ⟨0, 1, 1⟩⟩, ⟨45/2, ⟨0, 0, 2⟩⟩,
   ⟨-5, ⟨0, 1, 0⟩⟩, ⟨895/28, ⟨0, 0, 1⟩⟩, ⟨-57/2, ⟨0, 0, 0⟩⟩⟩,
  ⟨⟨1, ⟨2, 1, 0⟩⟩, ⟨-30, ⟨1, 0, 1⟩⟩, ⟨2395/294, ⟨0, 2, 0⟩⟩, ⟨-15/2, ⟨0, 1, 1⟩⟩,
   ⟨450/7, ⟨1, 0, 0⟩⟩, ⟨-3375/49, ⟨0, 1, 0⟩⟩, ⟨45, ⟨0, 0, 1⟩⟩⟩,
  ⟨⟨1, ⟨0, 1, 2⟩⟩, ⟨2/5, ⟨2, 0, 0⟩⟩, ⟨-3/7, ⟨1, 1, 0⟩⟩, ⟨114/5, ⟨1, 0, 1⟩⟩,
   ⟨-4/3, ⟨0, 1, 0⟩⟩, ⟨-3, ⟨0, 0, 1⟩⟩, ⟨-38/5, ⟨0, 0, 0⟩⟩⟩,
  ⟨⟨1, ⟨1, 2, 0⟩⟩, ⟨28/5, ⟨2, 0, 0⟩⟩, ⟨-6, ⟨1, 1, 0⟩⟩, ⟨-14/15, ⟨0, 1, 0⟩⟩⟩,
  ⟨⟨1, ⟨0, 2, 1⟩⟩, ⟨15/7, ⟨0, 2, 0⟩⟩, ⟨-6, ⟨0, 1, 1⟩⟩, ⟨12, ⟨1, 0, 0⟩⟩,
   ⟨-90/7, ⟨0, 1, 0⟩⟩⟩, ⟨⟨1, ⟨2, 0, 1⟩⟩, ⟨-1/6, ⟨0, 1, 1⟩⟩, ⟨-5/14, ⟨0, 1, 0⟩⟩⟩,
  ⟨⟨1, ⟨1, 1, 1⟩⟩, ⟨-6, ⟨1, 0, 1⟩⟩, ⟨2, ⟨0, 0, 0⟩⟩⟩⟩}
```

In this final experiment the speed-up factor is about 87.

11.7.4 Summary of Experiments

Table 11.1 summarizes some timing measurements of the algorithm **`rdGbBC12`**.

Task	Theorema	Compiled Theorema	Speed – up Factor
`rdGbBC12`[*Lex*, *2 Variables*, *3 Polynomials*]	0.75 *s*	0.02 *s*	38
`rdGbBC12`[*Lex*, *2 Variables*, *4 Polynomials*]	1.19 *s*	0.02 *s*	60
`rdGbBC12`[*Lex*, *3 Variables*, *3 Polynomials*]	10.67 *s*	0.17 *s*	63
`rdGbBC12`[*Lex*, *3 Variables*, *4 Polynomials*]	33.44 *s*	0.48 *s*	70
`rdGbBC12`[*Lex*, *4 Variables*, *3 Polynomials*]	27.25 *s*	0.39 *s*	70
`rdGbBC12`[*Deg*, *2 Variables*, *3 Polynomials*]	1.16 *s*	0.02 *s*	60
`rdGbBC12`[*Deg*, *2 Variables*, *4 Polynomials*]	1.67 *s*	0.03 *s*	56
`rdGbBC12`[*Deg*, *3 Variables*, *3 Polynomials*]	13.94 *s*	0.17 *s*	82
`rdGbBC12`[*Deg*, *3 Variables*, *4 Polynomials*]	20.02 *s*	0.22 *s*	91
`rdGbBC12`[*Deg*, *4 Variables*, *3 Polynomials*]	53.11 *s*	0.70 *s*	76

Table 11.1: Time Measurements of **`rdGbBC12`**

12 Interpolation of Univariate Polynomials

This chapter is about the interpolation of univariate polynomials by Neville's algorithm and is based on [Wind06]. We will define the functor **`UnivPoly`** and the algorithms **`NevilleP`** and **`Eval-NevilleP`**. After shortly explaining the corresponding Theorema code, we will compare the computing times of original Theorema and Java code created by the Theorema-Java Compiler.

12.1 The Functor `UnivPoly`

The functor **`UnivPoly`** takes a field **`K`** and returns the univariate polynomial ring over **`K`** which represents polynomials as the tuples of their coefficients.

$$
\begin{array}{l}
\textbf{Definition}\Big[\text{"Univariate Polynomial Functor"},\ \textbf{any}[\mathbf{K}],\\
\quad \textbf{UnivPoly}[\mathbf{K}] = \textbf{Functor}\Big[\mathbf{P},\ \textbf{any}[\mathbf{p},\mathbf{q},\mathbf{n},\mathbf{a}],\\[1em]
\qquad \dfrac{\mathbb{s} = \langle\rangle}{\underset{P}{\in}[p] \Leftrightarrow \left(\left(p = \left\langle \underset{K}{0}\right\rangle\right) \bigvee \left(\text{is-tuple}[p] \bigwedge |p| > 0 \bigwedge \underset{i=1,\ldots,|p|}{\forall} \underset{K}{\in}[p_i] \bigwedge p_{|p|} \neq \underset{K}{0}\right)\right)}\\
\qquad \underset{P}{0} = \left\langle \underset{K}{0}\right\rangle\\
\qquad \underset{P}{1} = \left\langle \underset{K}{1}\right\rangle\\
\qquad \underset{P}{\text{index}}[p] = \begin{cases} 1 & \Leftarrow \underset{j=1,\ldots,|p|}{\forall}\left(p_j = \underset{K}{0}\right)\\ \underset{i=1,\ldots,|p|}{æ}\left(\left(p_i \neq \underset{K}{0}\right) \bigwedge \underset{j=i+1,\ldots,|p|}{\forall}\left(p_j = \underset{K}{0}\right)\right) & \Leftarrow \text{otherwise}\end{cases}\\
\qquad \underset{P}{\text{deg}}[p] = |p| - 1\\
\qquad \underset{P}{\text{coef}}[p, n] = \begin{cases} p_{n+1} & \Leftarrow n \geq 0 \bigwedge n \leq \underset{P}{\text{deg}}[p]\\ \underset{K}{0} & \Leftarrow \text{otherwise}\end{cases}\\
\qquad \underset{P}{\text{canonic}}[p] = \left\langle p_i \underset{i=1,\ldots,\underset{P}{\text{index}}[p]}{|}\right\rangle\\
\qquad \underset{P}{\text{const}}[a] = \langle a\rangle\\
\qquad p \underset{P}{+} q = \underset{P}{\text{canonic}}\left[\left\langle \underset{P}{\text{coef}}[p,i] \underset{K}{+} \underset{P}{\text{coef}}[q,i] \underset{i=0,\ldots,\text{Max}\left[\underset{P}{\text{deg}}[p],\underset{P}{\text{deg}}[q]\right]}{|}\right\rangle\right]\\
\qquad p \underset{P}{-} q = \underset{P}{\text{canonic}}\left[\left\langle \underset{P}{\text{coef}}[p,i] \underset{K}{-} \underset{P}{\text{coef}}[q,i] \underset{i=0,\ldots,\text{Max}\left[\underset{P}{\text{deg}}[p],\underset{P}{\text{deg}}[q]\right]}{|}\right\rangle\right]
\end{array}
$$

```
    $p \underset{P}{*} q = \underset{P}{\mathrm{canonic}}\left[\left\langle \sum^{K}_{j=0,\ldots,i} \underset{P}{\mathrm{coef}}[p, j] \underset{K}{*} \underset{P}{\mathrm{coef}}[q, i - j] \;\Big|_{i=0,\ldots,\underset{P}{\mathrm{deg}}[p]+\underset{P}{\mathrm{deg}}[q]} \right\rangle\right]$

    $a \underset{P}{\cdot} p = \underset{P}{\mathrm{canonic}}\left[\left\langle a \underset{K}{*} \underset{P}{\mathrm{coef}}[p, i] \;\Big|_{i=0,\ldots,\underset{P}{\mathrm{deg}}[p]} \right\rangle\right]$

    $p \underset{P}{/} a = \left\langle \underset{P}{\mathrm{coef}}[p, i] \underset{K}{/} a \;\Big|_{i=0,\ldots,\underset{P}{\mathrm{deg}}[p]} \right\rangle$

    $\underset{P}{\mathrm{eval}}[p, a] = \sum^{K}_{i=0,\ldots,\underset{P}{\mathrm{deg}}[p]} \underset{P}{\mathrm{coef}}[p, i] \underset{K}{*} a \underset{K}{\wedge} i$

      ]]
```

12.2 The Algorithms **NevilleP** and **Eval-NevilleP**

The algorithm **NevilleP** takes a list of data points (in the form of the tuples **x** and **a** of the same length **n**), a field **K**, and the univariate polynomial ring over **K** (returned by **UnivPoly**). It computes the Neville polynomial which is of degree **n-1** and goes through the given data points.

```
Algorithm["Neville", any[x, a, K, P],
 NevilleP[x, a, K, P] :=
   { $\underset{P}{\mathrm{const}}[a_1]$                                          ⇐ |x| = 1
   { where[n = |x|,                                        ⇐ otherwise
   {   $\left(\left\langle \underset{K}{-} x_1, 1\right\rangle \underset{P}{*} \mathrm{NevilleP}[x_{1\to|\square}, a_{1\to|\square}, K, P] \underset{P}{-}\right.$                         ]
   {      $\left.\left\langle \underset{K}{-} x_n, 1\right\rangle \underset{P}{*} \mathrm{NevilleP}[x_{n\to|\square}, a_{n\to|\square}, K, P]\right) \underset{P}{/} \left(x_n \underset{K}{-} x_1\right)$ ]
```

The algorithm **Eval-NevilleP** takes five parameters: the first four have the same meaning as those of **NevilleP**, the fifth one is an element **v** of **K**. **Eval-NevilleP** returns the value of the Neville polynomial at the given point **v**.

```
Algorithm["Neville Evaluation", any[x, a, K, P, v],
 Eval-NevilleP[x, a, K, P, v] :=
   { $a_1$                                                          ⇐ |x| = 1
   { where[n = |x|,                                                 ⇐ otherwise
   {   $\left(\left(v \underset{K}{-} x_1\right) \underset{K}{*} \mathrm{Eval\text{-}NevilleP}[x_{1\to|\square}, a_{1\to|\square}, K, P, v] \underset{K}{-}\right.$
   {      $\left.\left(v \underset{K}{-} x_n\right) \underset{K}{*} \mathrm{Eval\text{-}NevilleP}[x_{n\to|\square}, a_{n\to|\square}, K, P, v]\right) \underset{K}{/} \left(x_n \underset{K}{-} x_1\right)$ ]

   ]
```

```
Theory["Neville",
      Algorithm["Neville"]
 Algorithm["Neville Evaluation"]]
```

12.3 Compilation to Java

To create the domain of univariate polynomials over ℚ on the Java side, we execute

```
Java-DeclareDomain[UnivPolyQ = UnivPoly[ℚ],
 Definition["Univariate Polynomial Functor"]]
```

Additionally, we compile the theory "Neville":

```
Java-Theory2Java[Theory["Neville"]]
```

12.4 Timing Measurements

12.4.1 The First Experiment

We are now ready to perform some time measurements in both Theorema and Java.

```
Use[⟨Built-in["Tuples"], Built-in["Quantifiers"], Built-in["Numbers"],
  Built-in["Number Domains"], Built-in["Connectives"]⟩]
ComputationalSession[]
Use[⟨Definition["Univariate Polynomial Functor"], Theory["Neville"]⟩]
```

```
NevilleP[⟨1, 2, 3, 4, 5, 6, 7, 8, 9, 10⟩,
  ⟨3, 1, 5, 2, 6, 10, -1, -9, 15, 20⟩, ℚ, UnivPoly[ℚ]] // AbsoluteTiming
```

$$\left\{7.5937500, \left\langle -8, \frac{45533}{360}, \frac{-530407}{2016}, \frac{10340243}{45360}, \frac{-596971}{5760}, \frac{469523}{17280}, \frac{-2443}{576}, \frac{3353}{8640}, \frac{-773}{40320}, \frac{143}{362880} \right\rangle\right\}$$

```
EndComputationalSession[]
```

So, the computation of this Neville polynomial for 10 data points took Theorema about 7.59 seconds. Let us use now the compiled algorithm:

```
Java-UseTheories[{"Neville"}]
```

```
Java-UseDomains[{UnivPolyQ}]
```

```
Java-Compute[NevilleP[⟨1, 2, 3, 4, 5, 6, 7, 8, 9, 10⟩,
   ⟨3, 1, 5, 2, 6, 10, -1, -9, 15, 20⟩, ℚ, UnivPolyQ]] // AbsoluteTiming
```

```
{0.3125000, ⟨-8, 45 533/360, -530 407/2016, 10 340 243/45 360,
   -596 971/5760, 469 523/17 280, -2443/576, 3353/8640, -773/40 320, 143/362 880⟩}
```

The compiled Java code just needs about 0.31 seconds to compute this polynomial and, hence, it is about 25 times faster than the above computation performed in Theorema's computational session.

12.4.2 The Second Experiment

As a second experiment, we want to compute a Neville polynomial for 12 data points:

```
ComputationalSession[]
NevilleP[⟨1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31⟩,
  ⟨1/2, -3, 15, 7/9, 61, 0, -7, 2/13, 5, -2, -1/15, 20/29⟩,
  ℚ, UnivPoly[ℚ]] // AbsoluteTiming
EndComputationalSession[]
```

```
{31.6875000, ⟨500 106 151 202 507/872 862 842 880, -17 084 512 287 418 862 845 921/12 922 551 087 641 395 200,
   272 751 935 337 391 768 663/239 306 501 622 988 800, -1 286 857 066 635 570 953 603/2 584 510 217 528 279 040,
   135 129 780 087 013 929 611/1 076 879 257 303 449 600, -1 942 771 048 943 230 861/99 404 239 135 703 040,
   300 875 487 730 797 797/153 839 893 900 492 800, -825 438 116 711 096 071/6 461 275 543 820 697 600, 3 881 559 120 762 221/717 919 504 868 966 400,
   -1 843 141 983 702 389/12 922 551 087 641 395 200, 83 324 894 573/39 159 245 720 125 440, -16 081 636 153/1 174 777 371 603 763 200⟩}
```

```
Java-Compute[NevilleP[⟨1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31⟩, ⟨1/2, -3,
    15, 7/9, 61, 0, -7, 2/13, 5, -2, -1/15, 20/29⟩, ℚ, UnivPolyQ]] // AbsoluteTiming
```

```
{1.2968750, ⟨500 106 151 202 507/872 862 842 880, -17 084 512 287 418 862 845 921/12 922 551 087 641 395 200,
  272 751 935 337 391 768 663/239 306 501 622 988 800, -1 286 857 066 635 570 953 603/2 584 510 217 528 279 040,
  135 129 780 087 013 929 611/1 076 879 257 303 449 600, -1 942 771 048 943 230 861/99 404 239 135 703 040,
  300 875 487 730 797 797/153 839 893 900 492 800, -825 438 116 711 096 071/6 461 275 543 820 697 600, 3 881 559 120 762 221/717 919 504 868 966 400,
  -1 843 141 983 702 389/12 922 551 087 641 395 200, 83 324 894 573/39 159 245 720 125 440, -16 081 636 153/1 174 777 371 603 763 200⟩}
```

In this example the speed-up factor is again about 25.

12.4.3 The Third Experiment

In this example, we use the algorithm **Eval–NevilleP**:

```
ComputationalSession[]
Eval-NevilleP[⟨1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31⟩,
  ⟨1/2, -3, 15, 7/9, 61, 0, -7, 2/13, 5, -2, -1/15, 20/29⟩,
  ℚ, UnivPoly[ℚ], 181/13] // AbsoluteTiming
EndComputationalSession[]
```

```
{2.4531250, 12 319 766 785 038 848 315/1 240 483 244 261 378 364}
```

```
Java-Compute[Eval-NevilleP[⟨1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31⟩,
   ⟨1/2, -3, 15, 7/9, 61, 0, -7, 2/13, 5, -2, -1/15, 20/29⟩,
   ℚ, UnivPolyQ, 181/13]] // AbsoluteTiming
```

```
{0.3437500, 12 319 766 785 038 848 315/1 240 483 244 261 378 364}
```

The speed-up factor here is about 7.

12.4.4 The Fourth Experiment

Again, we use the algorithm **`Eval-NevilleP`** for a computation:

```
ComputationalSession[]
Eval-NevilleP[⟨1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 39, 41⟩,
  ⟨1/2, -3, 15, 7/9, 61, 0, -7, 2/13, 5, -2, -1/15, 20/29, 5/19, -7/91, 27/99⟩,
  ℚ, UnivPoly[ℚ], 181/13] // AbsoluteTiming
EndComputationalSession[]
```

```
{19.4531250, 13 767 650 491 180 189 006 940/97 430 965 333 210 375 499 061}
```

```
Java-Compute[
  Eval-NevilleP[⟨1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 39, 41⟩,
   ⟨1/2, -3, 15, 7/9, 61, 0, -7, 2/13, 5, -2, -1/15, 20/29, 5/19, -7/91, 27/99⟩,
   ℚ, UnivPolyQ, 181/13]] // AbsoluteTiming
```

```
{2.8593750, 13 767 650 491 180 189 006 940/97 430 965 333 210 375 499 061}
```

Also in this example the speed-up factor is about 7.

12.4.5 Summary of Experiments

Table 12.1 summarizes some timing measurements of the algorithm **`NevilleP`**.

Task	Theorema	Compiled Theorema	Speed – up Factor
`NevilleP`[6 *data points*]	0.44 *s*	0.02 *s*	22
`NevilleP`[8 *data points*]	1.84 *s*	0.09 *s*	20
`NevilleP`[*10 data points*]	7.63 *s*	0.33 *s*	23
`NevilleP`[*12 data points*]	31.02 *s*	1.3 *s*	23
`NevilleP`[*14 data points*]	124.98 *s*	5.27 *s*	24

Table 12.1: Time Measurements of **`NevilleP`**

Conclusion and Future Work

In this thesis we showed how to drastically speed-up the computation times of original Theorema programs by compiling them into executable Java byte code. The generated Java programs are not faster by a constant factor, but rather depends the achievable acceleration on a size-parameter.

However, the execution times of the compiled Theorema programs are still far away from handcoded Java or C programs. Hence, it is the major challenge for the future development of the Theorema-Java Compiler to come up with additional ideas and techniques to further increase the speed-up.

We have chosen Java as the target language of the compiler mainly because of the well supported J/Link and, secondly, because it is natural to believe that a object-oriented language should support the generic programming philosophy of Theorema, in particular the functor mechanism. However, it would still be reasonable to try out plain C as the target language because of its efficiency, since, in fact, the method which we use for compiling Theorema functors (substitution of concrete function calls for function variables in compiled code) does not depend on the availability of object oriented features.

References

[Buch65] B. Buchberger. An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal (German). Department of Mathematics, University of Innsbruck, Austria. PhD Thesis, 1965.

[Buch91] B. Buchberger. Groebner Bases in Mathematica: Enthusiasm and Frustration. In: *Proceedings of the IFIP Working Conference on Programming Environments for High-Level Symbolic Computation*, September 23-27 1991, Karlsruhe, Germany; P.W. Gaffney, E.N. Houstis (eds.); pp. 80-91.

[Buch96a] B. Buchberger. Symbolic Computation: Computer Algebra and Logic. In: *FroCoS: Frontiers of Combined Systems, Applied Logic Series*; F. Baader, K.U. Schulz (eds.); Kluwer Academic Press, 1996, pp. 193– 220.

[Buch96b] B. Buchberger. Mathematica as a Rewrite Language. In: *Functional and Logic Programming*, Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, November 1-4, 1996, Shonan Village Center; T. Ida, A. Ohori, M. Takeichi (eds.); pp. 1-13.

[Buch96c] B. Buchberger. Using Mathematica for Doing Simple Mathematical Proofs. In: *Proceedings of the 4th Mathematica Users' Conference*, Tokyo, November 2, 1996, pp. 80-96, Copyright: Wolfram Media Publishing.

[Buch96d] B. Buchberger. Symbolic Computation: Computer Algebra and Logic. In: *Frontiers of Combining Systems, Proceedings of FROCOS 1996 (1st International Workshop on Frontiers of Combining Systems)*, March 26-28, 1996, Munich; F. Bader, K.U. Schulz (eds.); Applied Logic Series Vol.3, 1996, Kluwer Academic Publisher, Dordrecht - Boston - London, The Netherlands, pp. 193-220.

[Buch96e] B. Buchberger. Mathematische Software-Systeme: Die Zukunft (Mathematical Software Systems: The Future). Informatik-Spektrum 19(2), 1996, Springer, Heidelberg, pp. 100-101.

[Buch97] B.Buchberger. Mathematica: A System for Doing Mathematics by Computer?. In: *Advances in the Design of Symbolic Computation Systems*. A. Miola, M. Temperini (eds.); 1997, Springer Vienna, ISSN 0943-853X, ISBN 3-211-82-844-3. RISC Book Series on Symbolic Computation, pp. 2-20.

[Buch98a] B. Buchberger. Introduction to Groebner Bases. In: *Gröbner Bases and Applications*; B. Buchberger, F. Winkler (eds.); London Mathematical Society Lecture Notes Series 251, 1998, Cambridge University Press, ISBN 0-521-63298-6, pp. 3-31.

[Buch98b] B. Buchberger. An Algorithmic Criterion for the Solvability of a System of Algebraic Equations. In: *Gröbner Bases and Applications*; B. Buchberger, F. Winkler (eds.); London Mathematical Society Lecture Notes Series 251, 1998, Cambridge University Press, ISBN 0-521-63298-6, pp. 535-545.

[Buch98c] B. Buchberger. Theorema: The Current Status. In: *Proceedings of the Second International Theorema Workshop*; B. Buchberger, T. Jebelean (eds.); pp. 5– 37.

[Buch99a] B. Buchberger. Theorema: A System for Supporting Mathematical Proving. Workshop of the Japanese Consortium on Formal Methods, Nagoya, Japan, 1999.

[Buch99b] B. Buchberger. Theorema: A Proving System Based on Mathematica. International Symposium on Symbolic and Numeric Scientific Computing, Research Institute for Symbolic Computation, Hagenberg, Austria, 1999.

[Buch99c] B. Buchberger. Theorem Proving Versus Theory Exploration. Invited Talk at Calculemus Workshop, University of Trento, Italy, July 11, 1999.

[Buch00] B.Buchberger. Theory Exploration with Theorema. Analele Universitatii Din Timisoara, Seria Matematica-Informatica XXXVIII(2), 2000. ISSN 1124-970X. Selected papers of the 2nd International Workshop on Symbolic and Numeric Algorithms in Scientific Computing, Oct. 4-6, 2000, Timisoara, Romania; T. Jebelean, V. Negru, A. Popovici (eds.), pp. 9-32.

[Buch03] B. Buchberger. Groebner Rings in Theorema: A Case Study in Functors and Categories. Johannes Kepler University Linz, Spezialforschungsbereich F013. Technical report no. 2003-49, SFB Report, November 2003.

[Buch04] B. Buchberger. Algorithm Supported Mathematical Theory Exploration: A Personal View and Stragegy. In: *Proceedings of AISC 2004 (7th International Conference on Artificial Intelligence and Symbolic Computation)*, 22-24 September 2004; B. Buchberger, J. Campbell (eds.); Springer Lecture Notes in Artificial Intelligence 3249. Copyright: Springer, Berlin-Heidelberg, RISC, Johannes Kepler University, Austria, ISSN 0302-9743, ISBN 3-540-232, pp. 236-250.

[Buch07a] B. Buchberger. Translation of Sequence Variables into Java. Personal communication, Theorema seminar, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2007.

[Buch07b] B. Buchberger. Translation of Functor Based Domain Definitions into Java. Personal communication, Theorema seminar, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2007.

[Buch08] B. Buchberger. Groebner Bases in Theorema Using Functors. Collection of Extended Abstracts; D.M. Wang (ed.); 1st International Conference "Symbolic Computation in Cryptography", Beihan University, Department of Computer Science, Bejing, April 28-30, 2008.

[BuLo82] B.Buchberger, R. Loos. Algebraic Simplification. In: *Computer Algebra - Symbolic and Algebraic Computation*; B. Buchberger, G. E. Collins, R. Loos (eds.); 1982, pp. 11-43. Copyright: Springer Verlag, Vienna - New York.

[BuWi98] B. Buchberger, W. Windsteiger. The Theorema Language: Implementing Object- and Meta-Level Usage of Symbols. In: *Proceedings of Calculemus 98, Eindhoven, Netherlands*, 1998.

[EFT92] H.D. Ebbinghaus, J. Flum, W. Thomas. Einführung in die mathematische Logik. BI Wissenschaftsverlag Mannheim/Leipzig/Wien/Zürich, 3.Edition, 1992. ISBN 3-411-15603-1.

[Gies07] M. Giese. Abstract Data Type Based Compilation of Theorema into Java. Personal communication, Theorema seminar, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2007.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addision-Wesley Professional Computing Series, 1994. ISBN 978-0201633610.

[Hibe95] C. Hiber. An Exploration of Improved Buchberger's Algorithms for the Construction of Gröbner Bases (German). Department of Computer Sciences, Saarland University, Germany. Diploma Thesis, 1995.

[Jebe07] T. Jebelean. Elimination of Pattern Matching. Personal communication, Theorema seminar, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2007.

[KuBu04] T. Kutsia, B. Buchberger. Predicate Logic with Sequence Variables and Sequence Function Symbols. In: *Proceedings of the 3rd International Conference on Mathematical Knowledge Management, MKM'04*; A. Asperti, G. Bancerek, A. Trybulec (eds.); Lecture Notes in Computer Science 3119, Sept 19-21, 2004, Springer Verlag, Bialowieza, Poland, ISBN 3-540-23029-7, pp. 205-219.

[Kuts07] T. Kutsia. Sequence Variables and Linear Pattern Matching. Personal communication, Theorema seminar, Research Institute for Symbolic Computation, Johannes Kepler University Linz, 2007.

[Maed97] R. Maeder. Programming in Mathematica. Addison-Wesley, 3rd edition, 1997. ISBN 0-201-85449-X.

[Mitc01] J.C. Mitchell, K. Apt. Concepts in Programming Languages. Cambridge University Press, 1st edition, 2001.ISBN 978-0521780988.

[Mma] Mathematica. Developed at Wolfram Research Inc., directed by S. Wolfram. http://www.wolfram.com.

[SrSt07] N. Sridranop, R. Stansifer. Higher-order Functional Programming and Wildcards in Java. In: *Proceedings of the 45th Annual Southeast Regional Conference*, Winston-Salem, NC, USA, March 23-24, 2007, ISBN 978-1-59593-629-5.

[Tma97] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, D. Vasaru. A Survey of the Theorema project. In: *Proceedings of ISSAC'97*, International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997; W. Kuechlin (ed.); ACM Press, ISBN 0-89791-875-4, pp. 384-391.

[Tma98] B. Buchberger, K. Aigner, C. Dupré, T. Jebelean, F. Kriftner, M. Marin, K. Nakagawa, O. Podisor, E. Tomuta, Y. Usenko, D. Vasaru, W. Windsteiger. Theorema: An Integrated System for Computation and Deduction in Natural Style. Technical report no. 98-25 in RISC Report Series, Johannes Kepler University Linz, Austria, December 1998.

[Tma99] B. Buchberger, C. Dupré, T. Jebelean, K. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger. Theorema: A Short Demo. In: *Proceedings of the International Mathematica Symposium '99*, Research Institute for Symbolic Computation, Austria, 1999.

[Tma00] B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger. The Theorema Project: A Progress Report. In: *Symbolic Computation and Automated Reasoning*, Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning; M. Kerber, M. Kohlhase (eds.); 6-7 August 2000, ISBN 1-56881-145-4, pp. 98-113.

[Tma00a] B. Buchberger, C. Dupré, T. Jebelean, K. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger. The Theorema Project: A Progress Report. In: *Proceedings of the 8th Symposium on the Integration of*

Symbolic Computation and Mechanized Reasoning, St. Andrews, Scotland, August 6-7 2000; M. Kerber, M. Kohlhase (eds.); pp. 100– 115.

[Tma00b] Theorema Group. Theorema: A System for Supporting Mathematical Proving Implemented in Mathematica. Conference of the Association for Symbolic Logic, Chicago, 2000.

[Tma06] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. Journal of Applied Logic 4(4), pp. 470-504. 2006. ISSN 1570-8683.

[Trot04] M. Trott. The Mathematica Guidebook: Programming. Springer, 2004. ISBN 0387942823.

[WiBu06] W. Windsteiger, B. Buchberger, M. Rosenkranz. Theorema. In: *The Seventeen Provers of the World*; F. Wiedijk (ed.); 2006, LNAI 3600, Springer Berlin Heidelberg New York, ISBN 3-540-30704-4, pp. 96-107.

[Wind99] W. Windsteiger. Building up Hierarchical Mathematical Domains Using Functors in Mathematica. In: *Electronic Notes in Theoretical Computer Science*; A. Armando, T. Jebelean (eds.); volume 23-3, Elsevier, 1999, pp. 83– 102.

[Wind01] W. Windsteiger. A Set Theory Prover in Theorema: Implementation and Practical Applications. Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria. PhD Thesis, 2001.

[Wind06] W. Windsteiger. Algorithmic Methods 1 (German). Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria. Lecture notes, winter semester 2006.

[Zapl04] A. Zapletal. Algorithms in Computer Algebra for Polynomial Ideals and Modules (German). Institute of Discrete Mathematics and Geometry, Technical University of Vienna, Austria. Diploma Thesis, 2004.

Curriculum Vitæ

Personal Data

Name	Alexander Zapletal
Nationality	Austria
Date and place of birth	July 18, 1979, Vienna, Austria
Email	alexander@zapletal.at

Education

1997	High school graduation at Gymnasium Neulandschule Laaerberg, Vienna.
1997-2004	Studies in computer science and mathematics at Technical University of Vienna, Austria.
2004	Diploma degree in computer science. Diploma thesis: "Algorithmen in der Computeralgebra für Polynomideale und -moduln" ("Algorithms in Computer Algebra for Polynomial Ideals and Modules").
2004-2008	Doctorate studies in symbolic computation at the Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria. Scientific Advisor: Prof. Bruno Buchberger.