Thesis submitted to the Graduate Division of the **Electrical Engeneering Faculty** of the **Federal University of Uberlândia** as a fulfillment of the requirements for the PhD degree in the area of **Artificial Intelligence** of the course of **Graduation in Electrical Engineering**.

José Romildo Malaquias

Computer Algebra in Modern Functional Languages

Prof. Dr. Antônio Eduardo Costa Pereira

Thesis Supervisor

Universidade Federal de Uberlândia

Uberlândia, MG

Brazil

February 26 2007

Dados Internacionais de Catalogação na Publicação (CIP)

M237c Malaquias, José Romildo, 1967-
Computer algebra in modern functional languages / José Romildo Malaquias. - 2007.
140 f. : il.

Orientador: Antonio Eduardo Costa Pereira.

Tese (doutorado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.
Inclui bibliografia.

1. Inteligência artificial - Teses. 2. Programação funcional (Computação) - Teses. I. Pereira, Antonio Eduardo Costa. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 681.3:007.52

Elaborado pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

Computer Algebra in Modern Functional Languages

José Romildo Malaquias

Composition of the Examining Board

Prof. Dr. Antônio Eduardo Costa Pereira	FEELT – UFU	Supervisor
Prof. Dr. Luciano Vieira Lima	FEELT – UFU	
Prof. Dr. Carlos Roberto Lopes	FACOM – UFU	
Prof. Dr. José Lopes de Siqueira Neto	DCC – UFMG	
Prof. Dr. Claudio Cesar de Sá	DCC – UDESC	

UFU

Contents

Listings

Resumo

Muitos sistemas de computação algébrica foram propostos e implementados. A maioria deles são implementados ou até mesmo implementam linguagens sem a propriedade da referência transparencial, o que torna difícil e até mesmo impraticável a prova de correção de programas. Esta tese apresenta um sistema de computação algébrica implementado como uma biblioteca na linguagem de programação Haskell, que é uma linguagem funcional moderna com a propriedade da referência transparencial desejada.

O autor apresenta os fundamentos e algoritmos básicos para manipulação de expressões algébricas em um contexto declarativo, compatível com a Matemática.

Examina-se a adequação de construções oferecidas pela linguagem Haskell para a implementação da biblioteca de uma forma modular, de forma que ela possa ser facilmente estendida com a inclusão de novas fórmulas algébricas e novas operações sobre fórmulas. Tais extensões devem ser compatíveis com versões anteriores da biblioteca.

Esta tese também contribui por mostrar que linguagens funcionais modernas como Haskell são viáveis para a programação de sistemas práticos, até mesmo podendo ser melhores que linguagens convencionais em alguns aspectos, como nível de abstração.

Palavras Chave

Inteligência Artificial, Computação Algébrica, Linguagens Funcionais, Programação Funcional, Haskell.

Abstract

Many computer algebra systems have already been proposed and implemented. Most of them are implemented in or even implement languages without the referential transparency property, making it difficult, if not impractical, to reason about algebra programs. This thesis presents a computer algebra system implemented as a library in the Haskell programming language, a modern functional language with the desired referential transparency property.

The author presents the foundations and basic algorithms for manipulation of algebraic expressions in a declarative context, compatible with Mathematics.

The adequacy of the constructs provided by the Haskell programming language is considered when implementing such a library in a modular way, so that it can be easily extended with the inclusion of new algebraic formulas and manipulations. Such extensions should keep compatibility with prior versions of the library.

This work also contributes for showing that modern functional languages like Haskell are viable for day to day programming, even beating conventional languages in some aspects, like level of abstraction.

Keyworkds

Artificial Intelligence, Computer Algebra, Functional Languages, Functional Programming, Haskell.

Acknownledgments

I am grateful to all the persons who, in different ways, have supported me in this project.

- First I am grateful to God for keeping me alive and for providing all the necessary conditions for the development of this work.
- I am grateful to my wife Neli, for the encouragement when needed, and for the love all the time.
- I am grateful to my children, Ana Carolina, Felipe e Luíza, who have understood why I have stayed so many hours in front of a computer screen, instead of being with them for more time.
- I am grateful to my parents, Dírcia Rosa and José Augusto, for bringing me to life and for teaching me how to be a good person.
- I am grateful to Prof. Antônio Eduardo Costa Pereira, who helped me in the most difficult momments when developing this work.
- I am grateful to all my friends and co-workers from the Computer Science Department at the Federal University of Ouro Preto, for the friendship and for the years I was allowed to be out doing this work.

José Romildo Malaquias

Chapter 1

Introduction

1.1 Computer Algebra

Computers have been used for numerical computations since its beginnings. However these general purpose machines can be used for transforming and combining symbolic expressions as well. That is, computers can be used not only to deal with numbers, but also with abstract symbols representing mathematical formulas. This fact has been realized much later and is only now gaining acceptance among mathematicians and engineers.

Winkler [1] draws a good short introduction to Computer Algebra. Before 1850 mathematicians solved the majority of their problems by extensive calculations. However in the 19th century the style of mathematical research changed from quantitative to qualitative aspects. The advent of modern digital computers in general and by the development of program systems in Computer Algebra, in particular. played a role on this shift. Although even in our days many mathematicians think that the role of computers is number crunching and their role is the application of the appropriate algebraic transformations to the problem, underestimating the power of computing systems for algebraic manipulation, already in 1844 Lady Augusta Ada Byron, countess of Lovelace, recognized that this division of labor is not inherent in mathematical problem solving. In describing the possible applications of the *Analytical Engine* developed by Charles Babbage she wrote [1]:

"Many persons who are not conversant with mathematical studies imagine that be-

> cause the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its process must consequently be arithmetical and numerical rather than algebraic and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraic notation were provisions made accordingly."

And indeed computer is a "universal" machine capable of carrying out an arbitrary algorithm, being it numerical, symbolic or of any other nature.

But what does one mean by Computer Algebra or symbolic algebra computation? R. Loos wrote[2]:

> "Computer Algebra is that part of computer science which designs, analyzes, implements, and applies algebraic algorithms."

Algebraic algorithms are algorithms that transforms algebraic formulas (built from numbers, variables and function applications) following the rules from Algebra, a field in the Mathematics.

So Computer Algebra systems deal not only with integers and reals, but also with variables representing unknown quantities and any expression combining numbers and variables by means of mathematical operations. Numbers are represented exactly. The approximations of floating point numbers for reals are not acceptable anymore. So there is no need to concern about aproximation errors.

Computer Algebra software find its applications in any field dealing with the formulation of *laws* in mathematical terms, using algebraic equations and/or analytic concepts such as ordinary and partial differential equations and integral theorems. After the formulation of a law it should be solved. Then enters the Computer Algebra to assist the scientist in that.

1.2 Motivation

Nowadays there are many programs which perform Computer Algebra. These said programs find applications in areas like Mathematics, Computer Sciences, Engineering, Economics, etc. With these programs one can perform algebraic simplifications and handle literals, as shown in the following examples.

$$5 \times \frac{a^2 - 2ab + b^2}{a - b} = 5a + 5b$$
$$\frac{d}{dx}(ax^2 + bx + c) = 2ax + b$$
$$\sin^2 a + \cos^2 a = 1$$

Most of these programs suffer from the drawback that their programming languages are not referentially transparent, that is, they do not allow substitution of equals for equals, invalidating the use of certain Mathematical laws. Even if other areas of thought are able to do without substitution of equals for equals, this is not true of Mathematics, the main subject of Computer Algebra. Therefore, the absence of referential transparency is a mortal sin, when it occurs in a Computer Algebra language.

The notion of transparency is due to Whitehead and Russel [3]. Antoni Diller [4] illustrates the invalidation of substitution in natural languages with the following sentence drawn from Russell [5]:

> George IV wished to know whether Scott was the author of Waverley.

In chapter 3 the reader will find a full account of this discovery of Russell's. For the time being, it should be noticed that the above sentence is true, since the mad king really wished to know whether Scott was the author of Waverley. However, we do know that Scott and the author of Waverley are indeed the same person. Therefore, we may feel entitled to substitute Scott for *the author of Waverley* in Russell's sentence. In doing so, we get

> George IV wished to know whether Scott was Scott.

The result is obviously false, for even being mad, the King knew well that Scott was Scott.

Imperative computer languages suffer from the same lack of transparency as a natural language. For example, one cannot make use of the commutative property of multiplication to conclude that the C program of listing 1.1 will print the same number twice, although $f(2) \times g(5) = g(5) \times f(2)$ in Mathematics. This happens in C because C functions may change the state of the computing system (by means of assignments to variables) besides (maybe) computing a value. In program 1.1 function `g` changes the contents of the global variable `k` before returning a value and function `f` needs the value of `k` in order to compute a result value. So calls to `g` has an effect on calls to `f`. The return value of `f` depends on the previous calls to `g`. This program outputs `-6=99`.

Listing 1.1: An opaque C program.

```
#include <stdio.h>

int k = 1;

int g(int x)
{ k = k + x;
  return k;
}

int f(int x)
{ return k - x;
}

void main()
{ printf("%d=%d\n", f(x)*g(x), g(x)*f(x));
}
```

So it is highly desirable that the language used to express the Computer Algebra algorithms be referentially transparent.

1.3 Objectives

As has been noted, current Computer Algebra systems are implemented and/or support imperative programming languages, which are based on state changing and consequently lack referential transparency. The author proposes to develop a new Computer Algebra system withouth this restriction, as a library in the *Haskell* language, a modern functional language without side

effects.

The purposes of this work are:

- to provide the basics for implementing a Computer Algebra system embeded in a modern functional language, by implementing a small library of data types and functions for the manipulation of algebraic formulas;
- to examine the adequacy of the Haskell language for the construction of a modular and easily extendable library;
- to provide fundamental algorithms for manipulation of algebraic expressions in a declarative context, compatible with Mathematics, that uses transparent languages.

Currently there is plenty of Computer Algebra systems, as the reader can find in section 1.5. Many of them are comercial products that resulted from many years of work of many skilled developers. Those systems are very broad and complex, and find applications in many areas and are too broad and complex, being already stablished tools for professional development. It is not the intent of this work to provide a system that competes with them nor to provide a complete package, since this would require many hours of several dedicated programmers. In a *first step*, we hope to provide instead the implementation of a small library that can be used for the development of such system in the Haskell programming language.

A Computer Algebra system, due to its characteristics, is constructed in the following steps:

1. A group of people stablishes a core system capable of reading expressions of the domain and comparing them for equality. The comparison should be semantic and not syntatic, that is, expressions like $2 * y$ and $y + y$ are considered equal. The same group of people also stablishes a programming language. The core system built in this way is proposed to the comunity, together with simple examples like elementary integrations and derivatives, etc.

2. If the proposed system is accepted, specialists from many areas of knowledge that require symbolic manipulation start to add packages to the core system: more complex integrals, differential equations, topology, tensorial calculus, series, etc. The amount of knowledge

is so broad that it is not feasible for the group who started the system to develop all the packages. Also they would not have all the knowledge and ability to take this task.

A final user of a Computer Algebra system reading this text should keep in mind that it describes only a starting core system, which is implemented by making use of some inovative techniques related to the implementation language. Therefore users of Macsyma or Mathematica (to mention a few successfull products) should not expect a description in this text of a Computer Algebra system of the same level of those products, which are really products for the final user. What is described here is only a core system embeded in a modern programming language, which is not ready for the users, but should be further developed by the comunity if accepted by the comunity. Then after some years of development it could become a system as complete as Macsyma or Mathematica systems.

This project consist not only the design of a library in a given language, but also a study of language constructs offered or even missing in the programming language in face of the needs of the library. To accomplish this task some advanced knowledge of type systems and language constructs that goes beyound undergraduate level are needed.

In a *second step*, the library will be reimplemented in the *Clean* programming language[6], another modern functional language with a very good compiler, which will bring more efficiency to the system. More specific algorithms, like derivatives, integrals, differential equations, vectors and trigonometry, will also be implemented, making it suitable for application on real world problems.

1.4 Features of the Haskell Language

The Haskell programming language have the following features:

- It is a modern functional language.
- It is strongly typed.
- It does type inference.

- Its type system is polymorphic, with
  - parametric polymorphism and
  - overloading polymorphism.
- A program is made up of a set of algebraic data type definitions and functions.
- There are high order functions.
- Functions can be defined by pattern matching.
- Lazy evaluation of expressions.

1.5 History

According to Nilsson [7], research in Computer Algebra had its beginnings when James Slagle created the program SAINT, that was able to integrate functions by elementary methods. After this first step, Joel Moses improved the algorithms of SAINT, creating another program of symbolic integration which was named SIN.

In the sixties, Carl Engleman, Joel Moses and William Martin (at the Massachusetts Institute of Technology, as part of the MAC project) started the project **Macsyma**[1] [8], which is one of the finest systems of Symbolic Mathematics. It is quite large, and written in Lisp. After Macsyma, many other systems of Computer Algebra came to light. Each of these systems tried to correct a real or imaginary weakness of Maclisp, the original computer language of Macsyma. A few of these systems are:

- **Reduce**[2]. In the beginning, Lisp had many dialects. In general, every machine had its own dialect. Macsyma was developed in Maclisp. Therefore, one could not build the system in a machine running Franz Lisp, Interlisp, etc. The designers of Reduce [9][10] proposed a standard dialect of Lisp, which should be portable to any machine. Although Reduce became very popular, Standard Lisp was badly beaten by Common Lisp in the

[1] http://www.macsyma.com/
[2] http://www.rrz.uni-koeln.de/REDUCE

struggle to become the standard dialect of the main AI language. In the mean time, Macsyma migrated from MACLISP to Common Lisp. The result is that Macsyma runs in a standard language, while Reduce is built on top of an obscure dialect known as Standard Lisp.

- **Maple**[3]. Lisp always had a bad reputation for being slow. Therefore, Maple [11] was developed (at the University of Waterloo, Canada) in a procedural language (C) with a view to greater efficiency. Since procedural languages like C are not fit for symbolic computation, the implementors of Maple were forced to invent a symbolic language. Not being experts in compiler construction, they wrote an interpreter for the Maple language. The result is that Maple is much slower than Macsyma and other Lisp based systems. Maple's source code is not in the public domain.

- **Derive**[4]. This package (and its precursor **MuMATH**) was designed to offer a small and fast Computer Algebra system [12]. It was the only competitor to Macsyma that really delivered its promises. The system is indeed very small and reasonably fast. However, it did not meet the commercial success of Reduce and Maple.

Maple, Reduce and Derive were designed to be worthy competitors of Macsyma. There are also systems designed to offer limited functionality in a friendly environment. Among these systems, **MatLab**[5] [13] and **Mathematica**[6] [14] became imensily popular.

It is interesting to note that these two systems that promised limited functionality failed to deliver their promise too. Due to demands from the market, the developers of both MatLab and Mathematica increased the functionality of their systems, which became as fat as Reduce. However, neither MatLab nor Mathematica have the well designed architecture of Reduce. In fact, these system suffered from a chaotic growth.

Other systems are:

- **Magma**[7]. Magma is a large software package designed to solve computationally hard

[3] http://www.maplesoft.com/
[4] http://www.derive.com/
[5] http://www.mathworks.com/
[6] http://www.mathematica.com/
[7] http://www.maths.usyd.edu.au/u/magma/

problems in algebra, number theory, geometry and combinatorics. It provides a mathematically rigorous environment for computing with algebraic, number-theoretic, combinatoric and geometric objects. Magma's language is imperative with standard imperative-style statements and procedures, but offers a functional subset providing closures, higher-order functions, and partial evaluation.

- **Axiom**[8]. Axiom was originally developed as a research tool by IBM in collaboration with experts around the world. The unique strength of AXIOM is derived from its object-oriented approach and its overall structure which is strongly typed and hierarchical. AXIOM's algebra library is built on Common LISP.

- **Yacas**[9]. Yacas, standing for **Yet Another Computer Algebra System**, is a small and highly flexible Computer Algebra system and language. It is written in C++ and can be embedded into other applications. Things implemented include arbitrary precision integer arithmetic, rationals, complex numbers, vectors, matrices, derivatives, Taylor series, equation solving. It is being developed by Ayal Pinkus towards a symbolic calculator and there are plans to have a user interface through an internet browser.

- **Ginac**[10]. Ginac is a C++ library for doing computer algebra. It allows symbolic manipulation from within a C++ program. It is currently in active development, and looks very promising. GiNaC is an iterated and recursive acronym for GiNaC is Not a CAS, where CAS stands for Computer Algebra System. It is designed to allow the creation of integrated systems that embed symbolic manipulations together with more established areas of computer science (like computation- intense numeric applications, graphical interfaces, etc.) under one roof.

- **HartMath**[11]. HartMath is a Computer Algebra system written in Java. All math functionality is written in Java itself. HartMath has roughly the same functionality as Yacas. Some of the main implemented features are big rational number arithmetic, symbolic

[8] http://www.iec.co.uk/symbolic_software.asp
[9] http://www.xs4all.nl/~apinkus/yacas.html
[10] http://www.ginac.de/
[11] http://home.t-online.de/home/khartlage/hartmath.htm

derivatives, linear algebra, plot functions, numeric functions, pattern matching rules and pure functions.

- **Jacal**[12]. JACAL is an interactive symbolic mathematics program. JACAL can manipulate and simplify equations, scalars, vectors, and matrices of single and multiple valued algebraic expressions containing numbers, variables, radicals, and algebraic differential, and holonomic functions. It is written in Scheme, a dialect of Lisp.

- **Maxima**[13]. Maxima is a Macsyma clone, licensed under the GPL (GNU General Public License). It can be found in the GNU repositories, and is written in Common Lisp. It seems there is currently no maintainer for Maxima.

- **MockMMA**[14]. MockMMA is a Mathematica-style parser and pattern matcher, written in Lisp, with some additional mathematical functionality. One can manipulate polynomials in several variables over the integers, rational functions, and a variety of other mathematical objects. Manipulations include simplification, differentiation, integration, evaluation, pattern matching, etc. It is implemented in Common Lisp.

1.6 Text Structure

This text is organized in six chapters.

Chatper one: Introduction Presents the motivation of this work and its objectives as well as a brief history on the subject.

Chapter two: Formulas Explains how the data is organized in abstract types that describe algebraic expressions or formulas.

Chapter three: Context Justifies the need of keeping a set of flags, called the context, which are checked while doing the simplification of an algebraic formula, and provides information to decide what kind of transformations should be applied.

[12] http://swissnet.ai.mit.edu/~jaffer/JACAL.html
[13] http://www.gnu.org/software/maxima/maxima.html
[14] http://www.cs.berkeley.edu/~fateman/

Chapter three: Addition Explains the fundamental algorithms of the system for addition.

Chapter four: Multiplication and Subtraction. Discusses the algorithms used in symbolic multiplication and the use of the multiplication by -1 to achieve the subtraction.

Chapter five: Exponentiation and Division Presents the algorithms used both in exponentiation and in division.

Chapter six: System Organization Presents the module structure of the system and the results of a benchmarking program.

Chapter seven: Conclusion and Future Work Presents the conclusion and suggestions for future works.

Chapter 2

Algebraic Formulas

In this chapter, the reader will find descriptions abstract representations of the algebraic expressions in our Computer Algebra library. The author will be specially concerned with the basic manipulations of the foresaid expressions, but he will also discuss the result of computations that may fail or succeed.

For the sake of clarity, the term *formula* will be used for *algebraic expression*, as *expression* is already used when discussing programming languages and denotes a construct of the language not necessarily related to Algebra. This avoids ambiguity in terminology.

2.1 Kinds of Formulas

Formulas are expressions that can be manipulated in Algebra. They can be added and multiplied. They can be squared. Trignometric transformations may be applied to them. They can be diferentiated and integrated. A large number of operations can be applied to formulas. Formulas may be composed by numbers and variables, which can be combined in different ways.

Formulas can be grouped, according to their structure, in several groups, including:

- **Integers**, corresponding to the integer numbers, as defined in axiomatic theories like Peano's axioms, Church's numbers, or in one of the many brands of the Set Theory. A few examples of this group are 416, 7453 and -3291.

- **Constants**, representing known mathematical entities. They are expressed by a symbol or

a name, like π, e (neperian number) and $\imath$ (imaginary unit). In general, a constant formula stands for a number without an exact representation in a given numeric system.

- **Variables**, corresponding closely to mathematical variables. Logicians call them literals more often than not. Examples of literals: x, y, a, b, α, β.

- **Applications**, which are formulas built from simpler formulas by applying a functional operator to its arguments, which are themselves formulas. Exemplos: $\sin x$, $\coth x$, $\ln x$, $\log_{10} x$, $x+2$, $-a$, π^2 and $5 \times \sin[3\pi(\alpha+2)]$. Applications may have different forms, depending on the operator. The basic arithmetic applications are

  - **Sum**, denoting the sum of two formulas. Examples are $a+b$ and $13+y\times 4^x$.
  - **Product**, denoting the product of two formulas. For example $4 \times x$ and $(a+b) \times (a+c)$.
  - **Power**, denoting the power of two formulas, as in x^2 and $(a+b)^{\frac{1}{2}}$.

  There is no need for special forms for differences and quotients: they can be expressed using the above formulas. Basicaly, for any formulas x and y,

$$\begin{aligned} x-y &= x+(-1)\times y \\ \frac{x}{y} &= x\times y^{-1} \end{aligned}$$

  Other common applications are

  - **Logarithm** of a formula in a given base. Examples are $\log_{10} y$, $\log_e(x+y)$ and $\log_b b^c$.
  - **Trigonometric formulas**, like sine, cosine and tangent of a formula. Examples: $\sin x$, $\cos a^2$, $\tan(x^2+y^2)$.

  There are many other possible forms of applications (hyperbolic formulas, derivatives, integrals, vectors, matrices, series, ...)

- **Indeterminates**, which are formulas whose value cannot be determined, like those obtained by dividing by zero. Examples are $3/0$, 0^0 and $\log_0 x$.

2.2 Trivial Representation of Formulas

The representation of formulas in the implementation language is tricky. One could simply use an algebraic data type to obtain the disjoint union of the relevant groups of formulas. Listing 2.1 is a first attempt in defining the type of formulas using this approach. It includes only the

Listing 2.1: A first attempt in building a type representation for formulas.

```
module Formula where

data Fn = Sum                       -- sum
        | Pro                       -- product
        | Pow                       -- power

data Formula = Int Integer          -- an integer
             | Cte String           -- a known constant
             | Var String           -- a literal
             | App Fn [Formula]     -- an operator application
             | Ind Formula          -- an indeterminate formula
```

most basic groups of formulas: integers, named constants, variables, sums, products, powers and indeterminates.

The declaration of an algebraic data type defines a possibly recursive type and the set of its value constructors, specifying the types of the arguments of each constructor. For instance, type `Formula` has the following value constructors: **`Int`**, `Cte`, `Var`, `App` and `Ind`. For example, the contructor `Cte` has a sole argument, of type **`String`**. Constructor `App` has two arguments of types `Fn` and `Formula`.

Application formulas have a commom representation: an operator followed by a list of arguments. The type of the operator, `Fn`, is simply an enumeration of all possible operators: in this case sum, product and power. Each argument is a formula.

Table 2.1 exemplifies mathematical formulas and their corresponding representation by a value of the data type declared in listing 2.1. In the representation, operators (that is, the value

Math	Haskell
831	`Int 831`
π	`Cte "pi"`
x	`Var "x"`
$4+x$	`Sum App [Int 4, Var "x"]`
$a \times x$	`App Pro [Var "a", Var "x"]`
$x^{2\times a}$	`App Pow [Var "x", App Pro [Int 2, Var "a"]]`
$x-y\times z$	`App Sum [Var "x", App Pro [Int (-1), Var "y", Var "z"]]`
$\pi/2$	`App Pro [Cte "pi", App Pow [Int 2, Int (-1)]]`
$m/0$	`Ind (App Pro [Var "m", App Pow [Int 0, Int (-1)]])`

Table 2.1: Representing mathematical formulas.

constructors) take a prefix form. This prefix form is somewhat harder to read, but it eases the handling of expressions.

Note that the mapping between the integer representation and the integer field is constrained only by the amount of memory available in the computer, as Hasekll's **`Integer`** type represents arbitrary precision integers: its values can be as large as there is room for storing them in memory. There is no fixed number of bits for representing them as it happens in conventional languages like C, Pascal and Java.

One may deploy a constructor and its arguments as a pattern on the left hand side of equations used to define functions in Haskell, as in any other language that uses Landin's notation. Listing 2.2 shows an example of using a constructor in a pattern. The patterns of listing 2.2 play the role of selectors: they select components of a value of type `Formula`. In this example the selected component is one of the contructor arguments. Of course, this constructors can also be used to build a value of the corresponding type. This is shown in listing 2.3.

A naive addition operation is defined in listing 2.4. It is based in the following rules from

Listing 2.2: Selectors.

```
module Main where

import Formula

getName :: Formula → String
getName (Int i)       = show i
getName (Cte name)    = name
getName (Var v)       = v
getName (App fn args) = fnToString fn
    where
    fnToString Sum = "+"
    fnToString Pro = "*"
    fnToString Pow = "^"

main :: IO ()
main = print (getName (Cte "pi"))
```

Listing 2.3: Building a value of a data type.

```
module Main where

import Formula

mkSum :: Formula → Formula → Formula
mkSum x y = App Sum [x,y]

main :: IO ()
main = print (mkSum (Cte "pi") (Int 23))
```

Mathematics:

$$
\begin{aligned}
m+n &= \text{the sum of } m \text{ and } n \quad m,n \in \mathbb{Z} \\
x+0 &= x \\
0+x &= x \\
(x+y)+z &= x+y+z \\
x+(y+z) &= x+y+z
\end{aligned}
$$

Listing 2.5 shows examples of application of the function `add` to some formulas.

The main drawback of the proposed representation for formulas is that it does not allow

Listing 2.4: Addition of two formulas

```
add :: Formula → Formula → Formula
add (Int m)       (Int n)       = Int (m + n)
add x             (Int 0)       = x
add (Int 0)       x             = x
add (App Sum xs) (App Sum ys) = App Sum (xs ++ ys)
add x             (App Sum ys) = App Sum (x:ys)
add (App Sum xs) y             = App Sum (y:xs)
add x             y             = App Sum [x,y]
```

Listing 2.5: Examples of addition of formulas

```
x1, x2, x3, x4, x5, x6 :: Formula
x1 = add (Int 81) (Int 12)       -- Int 93
x2 = add (Cte "pi") (Int 0)      -- Cte "pi"
x3 = add (Int 0) (Var "alfa") -- Var "alfa"
x4 = add (App Sum [Int 5, Var "x"])
         (App Sum [Cte "e", Var "y"]) -- App Sum [Int 5, Var "x", Cte "e", Var "y"]
x5 = add (Var "x")
         (App Sum [Int 7, Var "y"])   -- App Sum [Var "x", Int 7, Var "y"]
x6 = add (App Sum [Int 5, Var "x"])
         (Cte "pi")                   -- App Sum [Int 5, Var "x", Cte "pi"]
```

easy extension of the type `Formula` without affecting all programs alreading using this type. Certainly the library will not offer all possible groups of formulas and someone probably would like to extended it. For example, suppose the library does not deal with logarithms and the user wishes to add support for them (effectively extending the library). A new value constructor for logarithms should be added in the definition of type `Fn`. Also new rules may need to be added in the relevant function definitions. Listing 2.6 shows an updated definition for the type of operators. Listing 2.7 shows an extended addition operation capable of adding two logarithms

Listing 2.6: Adding logarithms to the formula representation.

```
data Fn = Sum                    -- sum
        | Pro                    -- product
        | Pow                    -- power
        | Log                    -- logarithms
```

in the same base, according to the following mathematical law.

$$\log_b x + \log_b y = \log_b (x \times y)$$

Listing 2.7: Addition od formulas, including derivatives.

```
add :: Formula → Formula → Formula
add (Int m)         (Int n)         = Int (m + n)
add x               (Int 0)         = x
add (Int 0)         x               = x
add (App Log x b1) (App Log y b2) = if b1 == b2
                                      then App Log [mul x y, y1]
                                      else App Sum [App Log [x,b1], App Log [y,b2]]
add (App Sum xs)    (App Sum ys)    = App Sum (xs ++ ys)
add x               (App Sum ys)    = App Sum (x:ys)
add (App Sum xs)    y               = App Sum (y:xs)
add x               y               = App Sum [x,y]

mul :: Formula → Formula → Formula
mul = ...
```

Many kinds of formulas and the related operations may be implemented in the library: sums, products, powers, logarithms, trigonometric formulas (sine, cosine, tangent, etc.), inverse trigonometric formulas (arc sine, arc cosine, etc.), hyperbolic formulas, vectors, matrices, equations, derivatives and integrals, among others. Extending the type of formulas with a new constuctor for each new kind of formula will force the extension of the functions that manipulate formulas. The source code of the library would have to be changed in many points to accomplish that, and the level of modularization would be less then acceptable. So an alternative representation for formulas which make the library more modular is desirable.

Haskell 98, the current definition of Haskell, does not easily support data type extension with new constructors and corresponding extensions of functions defined on this type with new rules. Nonetheless most of Haskell implementations extends the language with new features that are useful in solving this problem: instance overlapping and multiparameter type classes. These and other common extensions to Haskell will be extensively used in the library. There is hope that most of these extensions will be integrated in the next language revision.

2.3 Extensible Union Types

Subtyping is a relation between two types which stablishes that every value of the subtype is also a value of the supertype. With the help of type classes we can introduce some kind of subtyping in Haskell, as is suggested by Liang, Hudak and Jones in [15].

We begin with a discussion of a key idea in our framework: how formulas may be expressed as extensible union types.

The disjoint union of two types is captured by the data type constructor **`Either`** of listing 2.8. It is a polimorphic data type constructor pre-defined in Haskell.

Listing 2.8: Disjoint union of two types.

```
data Either a b = Left a
                | Right b
```

Basicaly the type **`Either`** `a b` is the disjoint union of types `a` and `b`, where values of type `a` are labeled with **`Left`** and values of type `b` are labeled with **`Right`**. For example **`Left True`** and **`Right`** `'A'` are instances of the type **`Either Bool Char`**.

The union **`Either`** `a b` can be viewed as a supertype, and the summands types `a` and `b` as the subtypes.The operations that characterize the subtype relationship are

- injection of a value of a subtype into the supertype, which is accomplished by applying one of the value constructors **`Left`** or **`Right`** to the original value, and

- projection of a value of the supertype into a subtype, which is done by pattern matching the value of the supertype with one of the patterns **`Left`** *x* or **`Right`** *x*; if the matching succeeds, then the projection also succeeds binding *x* to the projected value; otherwise the projection fails.

The projection may fail, as the target subtype may not be the correct type for the original value.

An extensible union may be arbitrarily nested to accomodate a set of the subtypes. For example, the union of the types **`Integer`**, **`Char`**, `[(`**`Char,String`**`)]` and **`Int`** →**`Bool`**, may be expressed by the type declaraction in listing 2.9.

Listing 2.9: Example of an extesible union type

```
type T = Either Integer
                (Either Char
                        (Either [(Char,String)]
                                (Int → Bool)))
```

The injections and projections discussed above only work if the exact structure of the union is known. In order to have a single pair of injection and projection functions to work on all such constructions, a type class is used to capture the summand/union type relationship, refered here as a subtype relationship. The subtyping relationship between two types `sub` and `sup` will be expressed making both `sub` and `sup` an instance of the class `SubType`, defined in listing 2.10, which introduces the functions `inj` and `prj`. The success or failure of the projection function

Listing 2.10: Subtyping relationship

```
class SubType sub sup where
   inj :: sub → sup        -- injection
   prj :: sup → Maybe sub  -- projection
```

is represented by a value of type **`Maybe`** `sub`:

- **`Just`** x for a successful projection resulting in x, and
- **`Nothing`** for a projection that fails.

The partial function **`fromJust`** of type **`Maybe`** `a` →`a` defined in the standard Haskell library, can be used to access a successful result. Section 2.9 further discusses the representation of success and failure.

Note that listing 2.10 is not valid Haskell 98 code, as `SubType` is a multiparameter type class. However most Haskell implementations extend the language to include this feature.

Every type `a` is a subtype of the type **`Either`** `a b`, as listing 2.11 shows. Also, every type `a` which is a subtype of type `b`, is also a subtype of the type **`Either`** `c b`.

The instance declarations appearing in listing 2.11 overlaps and are not allowed in Haskell 98. Again most Haskell implmentations extend the language to allow overlapping instances.

Now that we have the means to stablish the subtyping relationship between two types, we can redefine the data type for formulas using this mechanism.

Listing 2.11: Subtyping relationship

```
instance SubType a (Either a b) where
   inj           = Left
   prj (Left x)  = Just x
   prj _         = Nothing

instance SubType a b => SubType a (Either c b) where
   inj           = Right . inj
   prj (Right x) = prj x
   prj _         = Nothing
```

2.4 An Extensible Type Representation for Formulas

The representation introduced earlier for formulas will be kept, but the representation of operators of application formulas will be made an extensible union type, in order to easily accomodate new kinds of application formulas. Listing 2.12 redefines the `Fn` data type from listing 2.1 as an extensible union type. Each operator has its own type, distinct from the types of all other

Listing 2.12: Representing the groups of application formulas

```
newtype Sum = Sum
newtype Pro = Pro
newtype Pow = Pow

type Fn = Either Sum (Either Pro Pow)

instance SubType Pow Pow where
    inj = id
    prj = Just
```

operators. The type `Fn` is a supertype of all operator types.

Listing 2.13 reimplements addition of formulas using this new types. Note how the rules for specific kinds of applications being added are implemented: there is an overloaded function called `addApp` which should be defined for each possible kind of application. It has three arguments: the operator of the first formula, its arguments, and the second formula. This function may succeed returning the sum of the formulas, or it may fail. The instance for Sum knows how to sum two formulas where the first of them is a sum.

To extend the library with logarithms:

1. Define a new type for the operator of logarithms.

Listing 2.13: Addition of formulas

```
add :: Formula → Formula → Formula
add (Int m)     (Int n)     = Int (m + n)
add (Int 0)     x           = x
add x           (Int 0)     = x
add (App op xs) y
    | isJust u              = fromJust u
    where
    u = addApp op xs y
add x           (App op ys)
    | isJust u              = fromJust u
    where
    u = addApp op ys x
add x           y           = App Sum [x,y]

class AddApp a where
    addApp :: a → [Formula] → Formula → Maybe Formula
    addApp _ _ _ = Nothing

instance (AddApp a, AddApp b) => AddApp (Either a b) where
    addApp (Left x)  = addApp x
    addApp (Right x) = addApp x

instance AddApp Sum where
    addApp Sum xs y@(App op ys)
        | prj op == Just Sum = Just (App (inj Sum) (xs ++ ys))
    addOp Sum xs y           = Just (App (inj Sum) (xs ++ [y])

instance AddApp Pro

instance AddApp Pow
```

2. Extend the type of operators, redefining the `Fn` type.

3. Define versions of any overloaded functions for operators.

Listing 2.14 extends our small library with logarithms following the above steps.

Listing 2.14: Extending the library of logarithms

```
newtype Log = Log

type Formula = Either Log (Either Sum (Either Pro Pow))

instance AddApp Log where
    addApp Log [x1,b1] y@(App op [x2,b2])
        | prj op == Just Log &&
          b1 == b2              = Just (App (inj Log) [mul x1 x2,b1])
    addApp _   _       _        = Nothing
```

In the next section the author introduces auxiliary functions for manipulating formulas, turning them into abstract data types.

2.5 Formula Representation with Existentially Quantified Type Variables

An alternative for representing extensible data types and functions relies on existentially quantified type variables [16] and type classes. These combination leads to a style of programming with similarities with the object oriented paradigm. Existential type variables are not part of Haskell 98, although most Haskell implementations have it as an extension to the language.

An existential type variable is mentioned only in the value constructors and is not part of the type constructor, allowing the construction of values of the same declared type, based on arguments of unrelated types. Listing 2.15 shows an example of a data type declaration with existential type variables. The components of the values cannot be accessed outside of the scope of the type variable. This means that to have access to the components, functions have to be attached to the value.

Arbitrary contexts are allowed before the constructors in the data type declaration. This way

Listing 2.15: Existential type variables in data type declarations

```
data Item = forall a . MkItem a (a → Bool)

f :: Item → Bool
f (MkItem a f) = f a

x, y :: Item
x = MkItem 5 even
y = MkItem 'B' isUpper
z = MkItem False not

results :: [Bool]
results = [ f x, f y, f z ]
```

the functions defined by classes in the context can be used to handle the values. Listing 2.16 illustrates this technique.

Listing 2.16: Context with existential type variables

```
data Item = forall a . (Eq a) => MkItem a (a → Bool)

f :: Item → Item → Bool
f (MkItem a f) (MkItem b g)
    | a == b    = False
    | otherwise = f a || f b

x, y :: Item
x = MkItem 4 odd
y = MkItem 'a' isLower

result :: Boolean
result = f x y
```

In order to represent the formulas, each kind of formula has its own data type declaration and does not have to concern about how the other kinds of formulas are represented. Functions handling each kind of expression should be also defined, again without concerning with every other kind of formulas. These functions should be overloaded in order to keep the same interface for handling any kind of formulas. In our example of addition of basic formulas, this means that there should be a data type for integers, constants, variables, sums, another for products and another for powers, toghether with one function for adding integers, another for adding constants, and so on. Consider the class `CFormula` is made up of the set of overloaded functions

over the types of each kind of formula. The general type of a single formula can then be expressed with an existential type variable and a context restricting the variable to instances of class `CFormula`, as appears in listing 2.17.

Listing 2.17: The general type of a formula using existential an type variable

```
class ( Eq f
      , Show f
      , CInt f
      , CCte f
      , CSum f
      , CAdd f
      , ... {- classes for other formulas and operations -}
      ) => CFormula f where
    where
    isEqual :: f → Formula → Bool
    isEqual _ _ = False

data Formula = forall f . (CFormula f) => Formula f

instance Show Formula where
    showsPrec p (Formula f) = showsPrec p f

instance Eq Formula where
    (Formula x) == y = isEqual x y

instance CFormula Formula where
    isEqual (Formula f) = isEqual f
```

Each specific type of a formula should be an instance of class `CFormula`. In order to interact with formulas of different kinds (like adding an integer to another integer, or to a constant, or to a sum, and so on), each kind of formula may have corresponding type classes with overloaded functions specific for the operations related to the kind of formula.

Listings 2.18, 2.19 and 2.20 shows how some integers, constants and sums can be represented. Listing 2.21 has code for the addition operation.

To extend the library with a new kind formula or a new operation is done by

1. defining a new data type for the formula or a new function for the operation
2. defining a new type class for the formula or operation, with all relevant instance definitions; defaults may be used for the kinds of formulas without a specific behaviour related to new kind of formula or operation

Listing 2.18: Integer formulas

```
newtype FInt = FInt Integer
               deriving (Eq,Show)

mkInt :: Integer → Formula
mkInt = Formula . FInt

class CInt f where
    isInt  :: f → Bool
    intVal :: f → Integer
    --
    isInt _ = False
    intVal = error "bug: intVal"

instance CInt FInt where
    isInt _ = True
    intVal (FInt x) = x

instance CInt Formula where
    isInt (Formula x) = isInt x
    intVal (Formula x) = intVal x

instance (CFormula a) => CInt a

instance CFormula FInt where
    isEqual (FInt m) x = isInt x && intVal x == m

-- Some useful formulas

zero  = mkInt 0
one   = mkInt 1
two   = mkInt 2
three = mkInt 3
four  = mkInt 4
mOne  = mkInt (-1)

isZero x = isInt x && intVal x == 0
isOne x  = isInt x && intVal x == 1
isMOne x = isInt x && intVal x == -1
```

Listing 2.19: Constant formulas

```
newtype FCte = FCte String
               deriving (Eq,Show)

mkCte :: String → Formula
mkCte = Formula . FCte

class CCte f where
    isCte  :: f → Bool
    cteName :: f → String
    --
    isCte _ = False
    cteName = error "bug: cteName"

instance CCte FCte where
    isCte _ = True
    cteName (FCte x) = x

instance CCte Formula where
    isCte (Formula x) = isCte x
    cteName (Formula x) = cteName x

instance (CFormula a) => CCte a

instance CFormula FCte where
    isEqual (FCte n) x = isCte x && cteName x == n

-- Some useful formulas

pi = mkCte ":pi"
e  = mkCte ":e"
i  = mkCte ":i"

isPi x = isCte x && cteName x == ":pi"
isE  x = isCte x && cteName x == ":e"
isI  x = isCte x && cteName x == ":i"
```

Listing 2.20: Sums

```
newtype FSum = FSum [Formula]
               deriving (Eq,Show)

mkSum :: [Formula] → Formula
mkSum = Formula . FSum

class CSum f where
    isSum  :: f → Bool
    terms :: f → [Formula]
    --
    isSum _ = False
    terms = error "bug: terms"

instance CSum FSum where
    isSum _ = True
    terms (FSum xs) = xs

instance CSum Formula where
    isSum (Formula x) = isSum x
    terms (Formula x) = terms x

instance (CFormula a) => CSum a

instance CFormula FSum where
    isEqual (FSum xs) y = isSum y && terms y == xs
```

Listing 2.21: Addition of formulas

```
add :: Formula → Formula → Formula
add x y
    | isJust u  = fromJust u
    | isJust v  = fromJust v
    | otherwise = mkSum [x,y]
    where
    u = addT x y
    v = addT y x

class CAdd a where
    addT :: a → Formula → Maybe Formula
    addT _ _ = Nothing

instance CAdd FInt where
    addT (FInt 0) x           = Just x
    addT (FInt m) x | isInt x = Just (mkInt (m + intVal x))
    addT _        _           = Nothing

instance CAdd FSum where
    addT (FSum xs) y | isSum y   = Just (mkSum (xs ++ terms y))
                     | otherwise = Just (mkSum (xs ++ [y]))

instance CAdd Formula where
    addT (Formula x) = addT x

instance (CFormula a) => CAdd a
```

3. adding the new type classes to the context of the `CFormula` type class, making the new functions available to all formulas through the general `Formula` data type.

Prior examples already ilustrates this process.

This solution is also based on other Haskell extensions: multiparametric type classes, overlapping instance declarations and undecidable instance declaractions. Some Haskell implementations provide all of them.

In a modular design, each kind of formula or specific operation should be definied in its own module, making almost all of the modules of the library mutually recursive. The major difficult found with this are related to deficiencies in the module system implementation. No Haskell implementation correctly provides support for mutually recursive module definitions, although it is a feature of Haskell. This was the reason why this solution to data type and function extendibility was not adopted.

2.6 Abstract Data Types

Section 2.6 describes an algebraic data type that one can use to represent formulas. During the development of a project, the programmer often has to add constructors, or to modify the representation of a given type. When this happens, one must modify every single module which happens to use the representation. For instance, if somebody renames `Var` to `V`, an application which makes use of the old name must undergo the appropriate changes. Besides this, in a collective project, all participants would have to understand well the definition of every single component of the whole. This may decrease productivity, since programmers must worry not only about their own chores, but also with their collaborators decisions.

With abstract data types, one avoids the shortcomings of algebraic types by hiding the type representation and other implementation details. A set of constructor functions, selectors and predicates replace the constructor patterns of algebraic types. Haskell programmers should resort to the module system in order to implement abstract data types. Only identifiers listed in the export list of a module are visible outside of a module. Any other identifiers defined in the module are kept hidden, serving as auxiliary definitions for the exported identifiers. In

order to hide the data type representation, the constructors should not be exported in the module interface. This has the disadvantage of not being possible to define functions on the abstract data type in client modules by pattern matching. In the absence of contructors to serve as patterns, one cannot access the components of a structure through pattern matching. In fact, one cannot even recognize the structure. The solution for this drawback is to export functions, which play the role of constructors, predicates, and selectors.

The theory of abstract data types is quite old, from a time when pattern match and equational definitions didn't exist. Therefore, this otherwise powerful tool does not provide the handy resources of pattern match, and forces the programmer to adopt a style that is at once old fashioned and difficult to read. Nevertheless, the necessity of preserving contributions even after modifying type declarations forces us to adopt abstract data types, and to make sure that users of the library do not manipulate formulas directly, but only through exported functions. Such functions are predicates for checking the kind of a formula, constructors for building a new formula from its constituent parts, and selectors for accessing components of a formula. The type abstract hides the implementation details from the users and keeps future changes in the representation from propagating to contributed code.

2.7 Basic Functions over Formulas

2.7.1 Integer Formulas

In Section 2.1, we have seen that integer formulas correspond to the mathematical integers. The basic operations on integer formulas are:

`mkInt :: Int →Formula`

constructs an integer formula from an integer value;

`intVal :: Formula →Int`

selects the integer value from the integer formula, and

`isInt :: Formula →Bool`

checks whether a formula is an integer formula.

In listing 2.22, the reader will find the Haskell definition of these three functions. Some

Listing 2.22: Integer functions.

```
mkInt :: Integer → Formula
mkInt = Int

isInt :: Formula → Bool
isInt (Int _) = True
isInt _       = False

intVal :: Formula → Integer
intVal (Int x) = x

zero, one, two, three, four, mOne :: Formula
zero  = mkInt 0
one   = mkInt 1
two   = mkInt 2
three = mkInt 3
four  = mkInt 4
mOne  = mkInt (-1)

isZero, isOne, isMOne :: Formula → Bool
isZero (Int 0) = True
isZero _       = False

isOne (Int 1) = True
isOne _       = False

isMOne (Int (-1)) = True
isMOne _          = False
```

useful integers like zero and one are defined at this point, together with predicates for them.

2.7.2 Constant Formulas

Constant formulas correspond to special entities found in Mathematics for which there is no direct and exact representation in the commonly used number systems. Examples of constants are π, e (neperian number) and $\imath$ (imaginary unit). Listing 2.23 shows the definitions of the functions used to handle constants. These functions are:

`mkCte :: String →Formula`

constructs a constant formula from the name that identifies it;

`cteName :: Formula →String`

selects the name of the constant formula, and

`isCte :: Formula →Bool`

checks whether a formula is a constant formula

Listing 2.23: Functions for handling constant values.

```
mkCte :: String → Formula
mkCte = Cte

isCte :: Formula → Bool
isCte (Cte _) = True
isCte _       = False

cteName :: Formula → String
cteName (Cte x) = x

pi, e, i, mI :: Formula
isPi, isE, isI, isMI :: Formula → Bool

pi = mkCte ":pi"
isPi (Cte ":pi") = True
isPi _           = False

e = mkCte ":e"
isE (Cte ":e") = True
isE _          = False

i = mkCte ":i"
isI (Cte ":i") = True
isI _          = False

mI = mkPro [mOne, i]
isMI x = isPro x && appArgs x == [mOne, i]
```

Some common constants are also defined, together with predicates for them.

2.7.3 Variable Formulas

Among other kinds of formulas, section 2.1 introduces the concept of a formula variable. It was said in the previous section that variable values correspond to mathematical variables. Therefore, they are primarily used to represent fixed, but unknown quantities. The basic constructors, selectors and predicates that one needs to work with variables are:

`mkVar :: String →Formula`

constructs a variable formula from a name that identifies the variable;

`varName :: Formula →String`

selects the name of the variable formula, and

`isVar :: Formula →Bool`

checks whether a formula is a variable formula.

In listing 2.24, the reader will find the definitions that must be added to the implementation module of the algebraic data types in order to handle formula variables.

Listing 2.24: Selectors, constructors and predicates for variables.

```
mkVar :: String → Formula
mkVar = Var

isVar :: Formula → Bool
isVar (Var _) = True
isVar _       = False

varName :: Formula → String
varName (Var x) = x
```

2.7.4 Indeterminate Formulas

Another kind of formula introduced in section 2.1 is the indeterminate formula. Such formula arises as the result of undefined operations, like division by zero, and do not represent any mathematical object. Corresponding constructor, selector, and predicate functions are defined for indeterminate formulas:

`mkInd :: Formula →Formula`

constructs an indeterminate formula from the original formula;

`indFor :: Formula →Formula`

selects the formula which is indeterminated;

`isInd :: Formula →Bool`

checks whether a formula is an indeterminate one

Listing 2.25, defines these functions.

Listing 2.25: Selectors, constructors and predicates for variables.

```
mkInd :: Formula → Formula
mkInd = Ind

isInd :: Formula → Bool
isInd (Ind _) = True
isInd _       = False

indFor :: Formula → Formula
indFor (Ind x) = x
```

2.7.5 Application Formulas

Representation

Integer, variable and constant formulas are atomic and cannot be decomposed into constituent parts. Application formulas are made of simpler formulas, combined by means of functions. They represent mathematical operations that have not been carried out (simplified), possibly because there is no way to simplify them or the user does not want it in a simplified form. Some basic application formulas are sums, products and powers, which are essentials to any treatment of applied Algebra.

Their representation is discussed in section 2.4. Basically, an application formula has an operator and a list of arguments. The type of the operator is an extensible union type. See listing 2.12.

The constructor, the two selectors, and the predicate associated to application formulas are given below. Their definitions are given in Listing 2.26.

`mkApp :: Fn →[Formula] →Formula`

constructs an application formula from an operator and a list of formulas (the arguments of the application);

`appFn :: Formula →Fn`

selects the operator of an application formula;

`appArgs :: Formula →[Formula]`

selects the arguments (list of formulas to which the operator is applied) of the application formula, and

`isApp :: Formula →Bool`

checks whether a formula is an application formula

Listing 2.26: Functions for handling applications.

```
mkApp :: Fn → [Formula] → Formula
mkApp = App

isApp :: Formula → Bool
isApp (App _ _) = True
isApp _         = False

appFn :: Formula → Fn
appFn (App fn _) = fn

appArgs :: Formula → [Formula]
appArgs (App _ args) = args
```

From listing 2.12, the reader can see that there are special operators for additions, products, powers, and for functions like sine, cosine and logarithm. However, there is no corresponding operator for the subtraction and division, since addition, product and power form a set of basic operations that can be used to represent these others. On the other hand, differences, quotients and roots are not basic, being thus expressed as sums, products and powers.

In the next section, special constructors, selectors and predicates for the basic arithmetic application formulas are discussed. Other applications, like logarithms, trigonometric functions and derivatives are similar.

Functions for basic operations

Besides the predicates and constructors for the four kinds of general formulas, the user will also need predicates to recognize the basic operations, to wit, addition, product, and power. These

functions, specifically designed for the basic operations, are described below. The reader can find their definitions in listing 2.27.

`mkSum :: [Formula] →Formula`

builds a sum from the terms of the sum;

`isSum :: Formula →Bool`

checks whether a formula is a sum;

`mkPro :: [Formula] →Formula`

builds a product from the factors of the product;

`isPro :: Formula →Bool`

checks whether a formula is a product;

`mkPow :: Formula →Formula →Formula`

builds a power from the base and the exponent of the power;

`isPow :: Formula →Bool`

checks whether a formula is a power;

`basePow :: Formula →Formula`

selects the base of a power formula

`exponentPow :: Formula →Formula`

selects the exponent of a power formula

A sum of some formulas (called terms) is an application of the addition operation to the terms. The corresponding function is `Sum`. There is a constructor and a predicate for sums. The terms of a sum may be selected using `appArgs`.

A product is an application of the multiplication operation to a list of formulas called factors. The corresponding function is *Prod*. There is a constructor and a predicate for products. The factors of a product may be selected using `appArgs`.

Listing 2.27: Constructors and selectors for basic applications.

```
-- Sums

mkSum []  = mkInt 0
mkSum [x] = x
mkSum xs  = mkApp (inj Sum) xs

isSum x = isApp x &&
          case prj (appFn x) of
            Just Sum → True
            _        → False

-- Products

mkPro [] = mkInt 1
mkPro [x] = x
mkPro xs = mkApp (inj Pro) xs

isPro x = isApp x &&
          case prj (appFn x) of
            Just Pro → True
            _        → False

-- Powers

mkPow x y = mkApp (inj Pow) [x,y]

isPow x = isApp x &&
          case prj (appFn x) of
            Just Pow → True
            _        → False

basePow :: Formula → Formula
basePow x
    | isPow x = head (appArgs x)

exponentPow :: Formula → Formula
exponentPow x
    | isPow x = head (tail (appArgs x))

half, pi2 :: Formula
half = mkPow two mOne
pi2 = mkPro [half,pi]
```

A power is an application of the power operation to two formulas, the base and the exponent of the power. The corresponding function if called `Pow`. There is a constructor, a predicate and two selectors for powers.

2.8 Canonical Form

The algorithms employed for the transformation of formulas consider that they are in a canonical form. This is important in order to have simpler algorithms.

Using the representation of formulas discussed in this chapter, the reader may notice that there is more than one possible representation for the same formula. See for example the formula

$$3+x$$

Although it is a simple formula, it can be represented as

```
App Sum [Int 3, Var "x"]
```

or as

```
App Sum [Var "x", Int 3]
```

due to the comutative law of addition. To avoid introducing additional complexity to the algorithms for algebraic transformations, a formula will be always represented uniquely, in what is called its **canonical form**. Every formula build by functions in the library will be in the canonical form. To achieve a canonical representation of formulas, an ordering among them should be stablished. The rules for this ordering follows.

Given to formulas x and y, x preceds y if and only if,

1. x is an integer and
   (a) y is an integer greater than x, or
   (b) y is not an integer
2. x is a constant and

   (a) y is a constant and the name of x lexicographicaly preceds the name of y, or

   (b) y is neither an integer nor a constant

3. x is a variable and

   (a) y is a variable and the name of x lexicographicaly preceds the name of y, or

   (b) y is neither an integer nor a constant nor a variable

4. x is an application formula with operator f_x and arguments a_x and

   (a) y is an application formula with operator f_y and arguments a_y and

      i. f_x precedes f_y or

      ii. $f_x = f_y$ and the a_x precedes a_y.

   (b) y is an indeterminate formula

5. x is an indeterminate formula encapsulating the formula x' and y is also an indeterminate formula encapsulating the formula y' and x' preceds y'.

The precedence of operators is given by the rules:

1. a sum does not precede any other operator

2. a product only precedes a sum

3. a power only precedes a product or a sum

4. any other operator precedes a power, a product or a sum

5. the precedence of non-sum, non-product and non-power operators is stablished, but irrelevant for the algorithms.

For example the ordering of terms in a sum is: first the integers (in increasing orders), than the constants (in increasing lexicographic order), than the variables (in increasing lexicagraphic order, then the operators other than sum, product or power, and then the powers finally followed by the products. The example above, $3 + x$, is thus represented by

```
App Sum [Int 3, Var "x"]
```

2.9 Representing Success and Failure

Some computations may succeed and produce a resulting value, or they may fail and no value is produced. This behaviour may be denoted using a parametric algebraic data type **Maybe** a for its result type, where a is the type of the successful value it produces. The **Maybe** a data type has two value constructors:

- one nullary constructor **Nothing :: Maybe** a for indication of failure and
- one constructor **Just** :: a →**Maybe** a of arity one for indication of success and whose argument is the resulting value.

The **Maybe** data type constructor is predefined as

```
data Maybe a = Just a
             | Nothing
```

Two other functions are defined for easy manipulation of **Maybe** values:

- a function **isJust :: Maybe** a →**Bool** that converts a **Maybe** value into a boolean value. Failure is converted to **False** and success is converted to **True**.

```
isJust :: Maybe a → Bool
isJust (Just x) = True
isJust Nothing  = False
```

- a partial function **fromJust :: Maybe** a →a to access the resulting value in the case of success. When applied to a failure value the result is undefined.

```
fromJust :: Maybe a → a
fromJust (Just x) = x
```

As an example, consider the computation of the real roots of the quadratic equation $ax^2 + bx + c = 0$, where a, b and c are real numbers. It is known that this equation has real roots only when the discriminant $b^2 - 4ac$ is not negative. In this case the computation fails. The

```
solve2 :: (Float, Float, Float) → Maybe (Float, Float)
solve2 (a, b, c)
    | d >= 0 = Just (r + s, r - s)
    | d <  0 = Nothing
    where
    d = b^2 - 4*a*c
    r = -b / (2*a)
    s = sqrt d / (2*a)
```

function `solve2` below has a triple formed by the coefficients of the equation as its input and a failure/success indication as its output. The function `useRoots` below tries to solve an equation when that is not possible it submits an error.

```
useRoots :: (Float, Float, Float) → Float
useRoots (a, b, c)
    | isJust roots = (x1 + x2)/(x1 - x2)
    | otherwise    = error "no␣real␣roots"
    where
    roots    = solve2 (a, b, c)
    (x1, x2) = fromJust roots
```

Chapter 3

Context

3.1 Formula Manipulation

In this work, arithmetic formulas are restricted to those formulas involving operations, such as addition, multiplication, exponentiation, logarithms, trigonometric functions, derivatives over algebraic values, and symbolic integration. These operations are expressed as functions in the implementation language. The application of a function may be written in prefix or infix (when binary) notation. Table 3.1 shows some of these functions.

operation	function name	notation	association	precedence	example
addition	`+`	infix	left	6	`x +y +z`
subtraction	`-`	infix	left	6	`x - y`
multiplication	`*`	infix	left	7	`x * y`
division	`/`	infix	left	7	`x / y`
exponentiation	`^`	infix	right	8	`x ^ y`
logarithm	**`log`**	prefix			**`log`** `x e`
natural logarithm	`ln`	prefix			`ln (x+y)`
sin	**`sin`**	prefix			**`sin`** `(`**`pi`**`*e^x)`
cos	**`cos`**	prefix			**`cos pi`**

Table 3.1: Some operations, where `x` and `y` are variables, and **`pi`** and `e` are constants.

The implementation language allows an identifier to be overloaded, which means that an identifier may name unambiguously more than one function within the same scope. Therefore, we reuse (with formulas) some names already bound in the implementation language standard

environment, like + and **sin**, for the addition operation and the sine function respectively. In Haskell, these two functions are defined for the builtin numeric types, i.e., for **Int** and **Float**. When the overloaded names have an infix notation, the association and precedence are defined in the implementation language standard environment. By doing so, we make sure that the notation used to express operations over formulas resembles the mathematical notation as far as possible, and the mathematical notation should be preferred in computational algebra systems. In this thesis the implementation language notation is used only when it illuminates the discussion, or simplifies handling of formulas by the machine. When feasible, the author prefers the pure mathematical notation over the implementation language notation.

A few examples of evaluation of formulas applying arithmetic operations follow.

$$2+\frac{3}{5}+\frac{8x^3}{x}-x^2 \Rightarrow \frac{13}{5}+7x^2 \tag{3.1}$$

$$a(a+b) \Rightarrow a^2+ab \tag{3.2}$$

$$\ln x^5 - \ln x \Rightarrow 4\ln x \tag{3.3}$$

$$\sin^2(x+y)+\cos^2(x+y) \Rightarrow 1 \tag{3.4}$$

The examples show that a computer algebra system is supposed to symplify any formula that cannot be reduced to a number. In general, a formula can be simplified in different ways, producing different results. For example, the formula

$$b^2 \cdot \frac{a}{b} + aa + bb + ab$$

can be simplified to result in

$$a^2 + 2ab + b^2$$

or in

$$(a+b)^2$$

This issue is investigated in the next section. For the time being, we will take a look at listing

3.1, which presents an example of using operations over formulas in a program. The example shows the symbolic solution of a quadratic equation. Note that symbolic variables are used exactly like numerical ones (like *integers* or *doubles*), making computation over formulas both easy and natural.

Listing 3.1: Solutions of the quadratic equation.

```
quadratic :: Formula → Formula → Formula → (Formula, Formula)
quadratic a b c = ( r + s, r - s )
    where
    r = - b / k
    s = (b^two - four * a * c)^(one/two) / k
    k = two * a
```

3.2 Controlling the Evaluation of Formulas

3.2.1 The Need of Controllers

An algorithm may need to evaluate a formula in many ways, depending on the desired form of the result. For example, the formula

$$b^2 \cdot \frac{a}{b} + aa + bb + ab$$

can be simplified to yield

$$a^2 + 2ab + b^2$$

or

$$(a+b)^2$$

depending on whether we need to expand or to factor it. So the algorithms discussed in this thesis would produce different results. Therefore, when invoking an algorithm, one must state which kind of simplification is to be carried out. This is done by means of *controllers*, i.e., identifiers associated to values that control evaluation of formulas. The system may check these controllers during the application of functions, and build the result of the function application

according to the values of the controllers. The set of all controllers forms the *context* in which formulas are calculated.

3.2.2 Representing Contexts

The contexts are states represented as records in the implementation language. Each controller is a field in the record. The value of a controller is the value of the corresponding field. Listing 3.2 shows an example of context. This initial context is bound to a global variable, since it may be used whenever one needs to restart a chain of state changes.

From listing 3.2, the reader can see that the notation to access a given controller is function application, to wit:

```
controller context
```

At the beginning of this section, it was said that context is a state representation. This means that the programmer may change one or more fields of the initial context. If we were dealing with a procedural language, or even with a functional language with mutable data structures, we would change the global context directly. Haskell, as well as other purely functional languages, does not allow this, in order to guarantee referential transparency. However, one may pass the context as argument to a function and change it in the context of a new call. Listing 3.3 gives an example of a function which changes the context. From that listing, the reader can see that the syntax for changing a set of selcted fields, while maintaining the others, has the following form:

```
context { controller1 =value1, controller2 =value, ... }
```

From listing 3.3, it is clear that the initial context binds default values to the controllers. The user can change these default values whenever a change of state is needed.

3.2.3 The Meaning of Some Controllers

The algebraic controllers described in this section enable the user of the library to have fine control over the transformations used to simplify an expression.

Listing 3.2: One may represent contexts as records.

```
data Context = Context { num_num   :: Int
                       , den_den   :: Int
                       , num_den   :: Int
                       , den_num   :: Int
                       , bas_exp   :: Int
                       , exp_bas   :: Int
                       , log_base  :: Formula
                       , zero_base :: Bool
                       , zero_exponent :: Bool
                       , pow_expd  :: Int
                       , log_expd  :: Int
                       , trig_expd :: Int
                       , branch    :: Bool
                       , bar       :: Bool
                       , point     :: Bool
                       }

iC :: Context
iC = Context { num_num       = 6
             , den_den       = 6
             , num_den       = 0
             , den_num       = 6
             , bas_exp       = -30
             , exp_bas       = 30
             , log_base      = e
             , zero_base     = False
             , zero_exponent = True
             , pow_expd      = 0
             , log_expd      = 0
             , trig_expd     = 0
             , branch        = True
             , bar           = False
             , point         = False
             }
```

Listing 3.3: State changing.

```
data Context = ...

iC :: Context
iC = ...

f1 x ctxt = f2 x (ctxt {den_num= 3})

f2 x ctxt = ... ctxt ...
```

The kinds of subformulas that are distributed over a formula or factored from a formula can be precisely controlled by setting an appropriate value for the appropriate controller. Positive integer values cause distribution whereas negative values cause factoring. The exact type of formula which is distributed or factored can be determined from table 3.2.

Prime	Type	Examples
2	numerical formulas	$4, -1/3, 5/7$
3	other nonsums	$x, \sin(x), z^3$
5	sums	$r+s, x^2-x, \ln x+z$

Table 3.2: Meaning of controllers.

If the value of the controllers `num_num`, `den_num`, `num_den`, `den_den`, `base_exp` or `exp_base` is an integer multiple of one of the primes listed in table 3.2. Then formulas of the type associated with that prime are distributed (or factored if the controller is negative) in accordance with the transformations associated with that control variable.

- `num_num`

  Default value: 6

  Controls the distribution (factoring) of factors in the numerator of a formula over (from) the terms of a sum in the numerator using the transformation:

  $$a \times (b+c) \equiv a \times b + a \times c$$

  Table 3.3 have an example of the use of `num_num` to control the distribution of factors over a sum. Note that differences are internally represented as sums involving negative coefficients.

- `den_den`

  Default value: 6

  Controls the distribution (factoring) of factors in the denominator of a formula over (from)

num_num	**result**
0	$3x \times (1+x)(1-x)$
2	$x \times (1+x)(3-3x)$
3	$3(1+x)(x-x^2)$
5	$3x \times (1+x-x \times (1+x))$
6	$(1+x)(3x-3x^2)$
10	$x \times (3+3x+x \times (-3-3x))$
15	$3(x-x^3)$
30	$3x-3x^3$

Table 3.3: Transforming $3x \times (1-x)(1+x)$ with different values for num_num

the terms of a sum in the denominator using the transformation:

$$\frac{1}{a \times (b+c)} \equiv \frac{1}{a \times b + a \times c}$$

For example, if den_den is 15, the formula

$$\frac{y}{x} \times \frac{1}{1+x} \times \frac{1}{1-x}$$

is transformed to

$$\frac{y}{x-x^3}$$

- den_num

  Default value: 6

  Controls the distribution (factoring) of factors in the denominator of a formula over (from) the terms of a sum in the numerator using the transformation:

$$\frac{b+c}{a} \equiv \frac{b}{a} + \frac{c}{a}$$

  For example, if den_num is 6, the formula

$$\frac{x+3}{\frac{3}{x}}$$

is transformed to

$$\frac{1}{3}+\frac{1}{x}$$

- num_den

  Default value: 0

  Controls the distribution (factoring) of factors in the numerator of a formula over (from) the terms of a sum in the denominator using the transformation:

$$\frac{a}{b+c} \equiv \frac{1}{\frac{b}{a}+\frac{c}{a}}$$

  This transformation yelds a kind of continuation-fraction expansion. For example, if num_den is 5, the formula

$$\frac{3+x}{1+x}$$

  is transformed to

$$\frac{1}{\frac{x}{3+x}+\frac{1}{3+x}}$$

  If num_den is 30, the result is

$$\frac{1}{\frac{1}{1+\frac{3}{x}}+\frac{1}{3+x}}$$

- bas_exp

  Default value: -30

  Controls the distribution (factoring) of the base of a formula over (from) the terms of an exponent which is a sum using the transformation:

$$a^{b+c} \equiv a^b \times a^c$$

  If bas_exp is 3, the formula

$$x^{1+y}$$

  is transformed to

$$x \times x^y$$

If `bas_exp` is a negative integer, common bases are collected under an exponent. Since this transformation is usually more desirable, the default value for `bas_exp` is -30.

- `exp_bas`

  Default value: 0

  Controls the distribution (factoring) of the exponent of a formula over (from) the base using the transformation:

$$(a \times b)^c \equiv a^c \times b^c$$

- `pow_expd`

  Default value: 0

  Controls the expansion of integer powers of sums in numerators and denominators. When it is a positive integer multiple of 2, then multinomial expansion occurs for positive exponents. When `pow_expd` is a positive integer multiple of 3, then multinomial expansion occurs for negative exponents. If the value of the `pow_expd` controller is 6, the formula

$$\frac{(1+x)^3}{(1+x+y)^2}$$

  yelds

$$\frac{1+3\times x+3\times x^2+x^3}{1+2\times x+2\times y+2\times x\times y+x^2+y^2}$$

- `branch`

  Default value: True

  Controls the application of the transformation

$$(a^b)^c \equiv a^{b\times c}$$

  when the formula c is not an integer. If c is a fraction, the transformation can effectively selects a particular branch (i.e. root) of a multiply-branched function. When `branch` is

false, the transformation is disable. For example, if `branch` is true,

$$(x^2)^{\frac{1}{2}}$$

yelds

$$x$$

If `branch` is false, the same formula is not transformed.

- `zero_exponent`

  Default value: True

  Controls the application of the transformation

  $$a^0 \equiv 1$$

  when the formula a is not a number. Since the transformation is valid for all a not equal to 0, the default value of `zero_exponent` is true. If a is a number, not equal to zero, the transformation is automatically applied irrespective of the vale of `zero_exponent`. If a is zero, the transformation results in an indeterminate formula.

- `zero_base`

  Default value: False

  Controls the application of the transformation

  $$0^a \equiv 0$$

  when the formula a is not a number. Since the transformation is not valid if a is zero or negative, the default value of `zero_base` is false. If a is a positive number, the transformation is automatically applied irrespective of the vale of `zero_base`. If a is zero or a negative number, the transformation results in an indeterminate formula.

3.2.4 Referential Transparency

In section 3.2.2, we have said that Haskell does not allow changing in a global value. This means that Haskell does not allow destructive updates of the value refered by a given variable. Although possible in other languages, destructive updating should be avoided, because it obscures the meaning of the program. To understand this, one must refer to books of Philosophy, where one discusses the concept of referential transparency. Bertrand Russel says that a language has referential transparency if one can substitute an expression's subexpression with another of equal meaning, preserving the meaning of the original expression. Since substitution is a fundamental operation in logic reasoning, a computer language must exhibit referential transparency if we ever want to reason about programs. An example from Russel will make clear what is referential transparency. Consider the following sentence:

> George IV wanted to know whether Scott was the author of Waverley.

Historians say that this sentence is true, i. e., George IV really wanted to know whether Scott had written Waverley. We know that Walter Scott was the author of Waverley. Therefore, the name *Scott* and the noun phrase *author of Waverley* are equivalent expressions, i.e., they refer to the same person. Therefore, if English is transparent, one can substitute *Scott* for *author of Waverley*. With this substitution, Russel's sentence becomes:

> George IV wanted to know whether Scott was Scott.

After the substitution, the sentence, which was true, becomes false. Of course, George IV did not want to know whether Scott was Scott. Even being mad, George IV knew perfectly well that Scott was indeed Scott. The replacement of an expression for its equivalent may falsify a true sentence! The conclusion we reach is that, if a language is not transparent, one cannot substitute equals for equals. Although acceptable in a natural language, this is undesirable in a programming language, because programmers must use substitutions to prove their code correct, or to perform source to source transformation. To understand this, consider the small program of listing 3.4. One cannot substitute `g(2)+h(5)` for `h(5)+g(2)`, even though the commutative property of the addition operation warranties this equivalence.

Listing 3.4: Procedural languages are not transparent.

```
program (input,output);
  var y = 5;

  function g(integer x): integer;
    begin
      y := 512;
      g := 2 * x
    end;

  function h(integer x): integer;
    begin
      h := x + y
    end;

  begin
    writeln(h(5)+ g (2))
  end.
```

3.2.5 Passing the Context as Input to Algorithms

It was said that the algorithms we are using make use of contexts to control simplification of formulas. Basically the controller's values are set with the desired values (the ones that will produce the desired result form) and then the simplification of formulas takes place in this customized context. The implementation of the algorithms may use the context in two ways:

- consult the value of some controller in the context and conduct simplification of the formulas according to the controllers value,
- create a new context on which formulas are simplified and their results are used to build the final result

Imperative programming languages are based on state changing. In such languages a variable is a placeholder that contains a value. The value is stored in the variable using assignment commands. Pascal, C, Java, Lisp, Scheme and ML are examples of imperative programming languages. In section 3.2.4, we have learned that this kind of destructive update leads to an undesirable situation, where reasoning about programs becomes very difficult. On the other hand, the use of contexts in procedural languages is trivial: the context may be stored as a value in a global variable. This variable could be accessed directly from any function that requires the

value of a controller. To produce a new context, first the original context should be stored in a temporary variable and then it can be changed as desired. Further simplification of formulas will use this modified context. When it is not needed anymore, the original context saved in the temporary variable is restored, and simplification can continue. This technique of saving and restoring the context is accomplished automatically in languages with dynamic scope for variables (like Lisp and some Scheme dialects).

Declarative languages do not support the concept of state changing and variables are just names for unknown, but otherwise fixed values. The values they represent cannot be changed. Logic and (pure) functional languages like Mercury, and Haskell are declarative. A solution to our problem would be to pass the context as an extra argument to the functions which manipulate formulas, as hinted in listing 3.3. If a new context is needed, it can be created and passed as an argument to the desired function. The original context is not changed.

As our implementation language is a pure functional programming language, we cannot expect to perform changes in a global context. So the algorithms should receive the context among their inputs. Thus if we need an algorithm to compute logarithms as a function named **`log`** of type

```
log :: Formula →Formula →Formula
```

that maps two formulas (the logarithmand and the base) to the corresponding logarithm, we will not be able to express this with only two arguments. After all, the function also needs the context for the algorithm that computes the logarithm. If `Context` is the type for contexts, the type of **`log`** should be

```
log :: Context →Formula →Formula →Formula
```

This means that **`log`** should map a context and two formulas to the logarithm. As an example, listing 3.5 contains the definition of a function that changes a controller in order to simplify its arguments.

Passing the context as an extra argument is an acceptable solution, but it has a drawback: the arity of each function will be incremented by one. For instance, the binary functions will become ternary. The deleterious consequence of this increase in arity is that one cannot overload

Listing 3.5: Context as argument.

```
f :: Context → Formula → Formula → Formula
f ctxt x y
    | isInt y   = log ctxt x y
    | isInt x   = log ctxt y x
    | isSum x   = log ctxt x + sin ctxt y
    | otherwise = g newCtxt (log ctxt x y)
    where
    newCtxt = ctxt { num_num = 0, den_den = 14 }

g ctxt x
    | num_num ctxt == 2 = x
    | otherwise         = expand ctxt x
```

operators belonging to Haskell's primitive environment. For example, there will be no way of writing `x+y` as the + function now needs a third argument.

A solution to this problem is to incorporate the context into the representation of formulas. Thus there will be no need of an extra argument as the context will be passed on together with the algebraic formulas, and the binary operators can be used without any problems. However, if we adopt this last solution, we will have to revise the represention of formulas to accomodate contexts. A drawback of this solution comes with functions with more than one formula as arguments: each formula comes packed with a context, so there is more than one context available to the function. How to choose one of them? This question has to be answered, as the formulas may have been created at different situations, in different contexts, possibly making each context distinct from the others. For this reason this solution will not be adopted[1].

3.3 Implicit Parameters

The context may be implicitely passsed as an additional parameter to the functions, simulating dynamic scope from other languages. This way the additional parameter does not have to be written explcitly in the function calls, allowing one to use infix notation as it is desired.

Implicit parameters [17] are a language feature that provides dynamically scoped variables within a statically typed Hindley-Milner framework. Implicit parameters are lexically distinct

[1] An earlier version of the library has adopted this solution. But later it was realized it is a poor solution and the author searched for better solutions for passing the context environment around.

from regular identifiers, and are bound by a special `with` construct whose scope is dynamic, rather static as with let. Implicit parameters are treated by the type system as parameters that are not explicitly declared, but are inferred from their use.

Although Haskell 98 does not provide implicit parameters, most Haskell implementations extend the language with this feautre, making it a feasable solution to the problem of passing the context to function calls without explicitly adding a new parameter to them. Thus, we can keep using binary operators to represent common binary arithmetic functions.

With implicit parameters, the example from listing 3.5 can be rewritten as in listing 3.6. The

Listing 3.6: Implicit context.

```
f :: (?ctxt :: Context) => Formula → Formula → Formula
f x y
    | isInt y   = log x y
    | isInt x   = log y x
    | isSum x   = log x + sin y
    | otherwise = g (log x y)
                  with ?ctxt = newCtxt
    where
    newCtxt = ctxt { num_num = 0, den_den = 14 }

g x
    | num_num ?ctxt == 2 = x
    | otherwise          = expand x
```

types of some functions used in this reworked example are

```
log    :: (?ctxt :: Context) => Formula → Formula → Formula
sin    :: (?ctxt :: Context) => Formula → Formula
expand :: (?ctxt :: Context) => Formula → Formula
g      :: (?ctxt :: Context) => Formula → Formula
```

3.4 Alternative Solutions for Passing a Context Around

An alternative solution is the generation of all possible simplifications for a formula. The list data structure, in conjunction with the lazy evaluation of Haskell can be used to generate a list of all simplifications, being each one calculated only if really needed. Wadler [18] discusses

backtracking in functional languages.

Another solution is to make the type of function arguments not formulas, but a more elaborate construct: functions having the context as input and the resulting formula as output. Then, the function would be called with some functional arguments which should be applied to a context, in order to find out the values on which to operate, and its result would be another function, also to be applied to an enviroment, to produce the desired result value. The definition would be something like the example from listing 3.7.

Listing 3.7: Passing contexts around.

```
:: (Context → Formula) → (Context → Formula) → (Context → Formula)
v1 v2 = h
  where h env | x == zero = y
              | y == zero = x
              | get env num_num == 5 =
                  -- the result depends on the context
                  g v1 v2 env
              | otherwise = (mul (add v1 v2) v1) env
             where x = v1 env  -- the value of the first arg
                   y = v2 env  -- the value of the second arg
```

The major problem with this solution is the loss of sharing among intermediate results [19]. To avoid that one has to make the functional values memoize their results, but this generaly imposes drastic performance penalties to the system. This is the case unless there is a good library for function memoization that could be implemented independently of the specific Haskell implementation being used.

Chapter 4

Addition

In this chapter the algorithms employed to find the sum of any formulas are presented.

4.1 Basic Addition Rules

The addition of two arbitrary formulas may be reductible or not, depending on the formulas being added. When it is reductible, the resulting sum is a formula computed by applying known rules to the terms of the addition. For example, the sum of two rational numbers is a rational number given by the rules for adding rational numbers, e.g.

$$\frac{3}{4}+\frac{5}{6}=\frac{\frac{12}{4}\times 3+\frac{12}{6}\times 5}{12}=\frac{19}{12}$$

But the addition may fail to produce a sum that is simpler than the stated addition. In this case there is no option but to pack the terms being added in an application formula for sums. For example, when adding an integer to a literal, e.g.

$$45+x$$

there is no way to simplify the resulting sum and the result is left as an indication of the desired addition. Sections 2.1 and 2.7.5 discuss the representation of unreductible formulas, like the one above.

The algorithms used to implement addition make use of a function that tries to add two terms producing a reduced formula when it is possible, or an indication of failure when not. This function handles the cases of addition to zero and addition of integers and rational numbers, with the proviso of handling special cases when necessary. Section 2.9 discusses the representation of success and failure by means of the **`Maybe`** data type.

4.2 Addition of Integer and Rational Numbers

The most basic rules handle addition of zero and any formula, and addition of integer and rational numbers.

Zero is the *identity* value of the addition operation, so that

$$0+x=x+0=x$$

for any formula x.

Integer addition is carried out following the arithmetic rules taught at elementary levels of Mathematics. The implementation makes use of the function

```
(+) :: Integer →Integer →Integer
```

from the standard library to handle it.

Mixed integer and rational addition is accomplished following the rule

$$n+\frac{p}{q}=\frac{n\times q+p}{q}$$

where n, p and q are integers and $q>0$.

Rational addition is carried out by following the rule

$$\frac{m}{n}+\frac{p}{q}=\frac{m\times\frac{k}{n}+p\times\frac{k}{q}}{k}$$

where

$$k = \mathrm{lcm}(n, q)$$

m, n, p and q are integer numbers, $n > 0$, $q > 0$ and $\mathrm{lcm}(n,q)$ is the least common multiple of n and q. In the most general case the result is a rational number. When $k = 1$, the result is an integer number.

Listing 4.1 shows the implementation code for the basic rules discussed in the above sections.

4.3 Handling of Special Cases

Besides the basic rules for addition, it is possible to specify special rules. For example, there is no provision for the addition of two logarithms in the core addition functions. How is it supposed to be achieved? The library offers a mechanism to attach special rules for addition into the algorithm that adds two formulas. The rules are triggered when none of the explicit rules shown above is adequate to handle the addition. For this scheme to work, at least one of the formulas being added must be an application formula.

The core addition algorithm has a hook for calling customized functions when none of the basic rules apply. The hook is an overloaded function called `sumT`, introduced by the class `SumC` defined in listing 4.2. `sumT` is a function with three explicit arguments:

1. the operator of the application formula (the formula corresponding to the first formula to be added),

2. the arguments of this application formula,

3. the second formula to be added.

The function may fail or succeed. If it succeeds the result is a formula corresponding to the sum of the two argument formulas. This listing also declares an instance **`Either`** `a b` of class `SumC`. This is important, as the type of the operator in application formulas is an extensible

Listing 4.1: Addition of two terms.

```
addTerms :: (?ctxt :: Context) => Formula → Formula → Maybe Formula
-- Try to add two terms using simple addition rules:
-- 1) zero addition: at least one of terms is zero
-- 2) integer addition: both terms are integers
-- 3) rational addition: both terms are rationals (integer or fraction)
-- 4) specific rules for the kinds of terms involved (via a table)
addTerms x y
    | isInt x              = addInt1
    | isInt y              = addInt2
    | isFrac x && isFrac y = addFracs
    | isApp x && isJust u  = u
    | isApp y && isJust v  = v
    | isInd x || isInd y   = Just (mkInd (mkSum [x,y]))
    | otherwise            = Nothing
    where
    u = sumT (appFn x) ?ctxt (argList x) y
    v = sumT (appFn y) ?ctxt (argList y) x
    addInt1
        | isInt y            = Just (mkInt (intVal x Prelude.+ intVal y))
        | isZero x           = Just y
        | isFrac y           = let nx      = intVal x
                                   (ny,dy) = numDenFrac y
                                   n       = nx Prelude.* dy Prelude.+ ny
                                   d       = dy
                               in  Just (mkRat n d)
        | isApp y && isJust v = v
        | otherwise          = Nothing
    addInt2
        | isZero y           = Just x
        | isFrac x           = let ny      = intVal y
                                   (nx,dx) = numDenFrac x
                                   n       = ny Prelude.* dx Prelude.+ nx
                                   d       = dx
                               in  Just (mkRat n d)
        | isApp x && isJust u = u
        | otherwise          = Nothing
    addFracs =
        let (nx,dx) = numDenFrac x
            (ny,dy) = numDenFrac y
            n       = nx Prelude.* (div d dx) Prelude.+
                      ny Prelude.* (div d dy)
            d       = lcm dx dy
            k       = gcd n d
        in  Just (mkRat (div n k) (div d k))
```

Listing 4.2: An overloaded function for special addition rules.

```
class SumC a where
    sumT :: (?ctxt :: Context) => a → [Formula] → Formula → Maybe Formula
    sumT _ _ _ _ = Nothing

instance (SumC a, SumC b) => SumC (Either a b) where
    sumT (Left x)  = sumT x
    sumT (Right x) = sumT x
```

union type, as discussed in sections 2.3 and 2.4. Applying `sumT` to a value of the union type is as simple as to apply the appropriate `sumT` function to the summand of the union.

The type of each possible operator of an application formula should be an instance of class `SumcC`, overloading the function `sumT` to work according to its own rules. Consider the addition of logarithms according to the rule

$$\log_b x + \log_b y = \log_b (x \times y)$$

Listing 4.3 implements this rule by making `Log`, the type of the operator for logarithm formulas, an instance of `SumC`. Basicaly the arguments of the logarithm are always the logarithmand (`x`)

Listing 4.3: Addition of logarithms.

```
instance SumC Log where
    sumT Log ctxt [x,b] y
        | isLog y && logBase y == b = Just (mkLog (x * logLog y) b)
        | otherwise                 = Nothing
```

and the base (`b`). The second formula taking part in the addition is `y`. It is checked to find if it is a logarithm with the same base as the first formula. If it is, then the rule can be applied and the function succeeds. Otherwise the function fails and it is not possible to apply the transformation to the addition of both formulas.

Other operations like multiplication 5 and exponentiation 6 have similar hooks for handling special rules.

Trying to keep the lenght of this dissertation not too long, not all special rules are presented. The interested reader can reach them at the source code, available as free software from the URL http://iceb.ufop.br/~romildo/calg/.

4.4 The Core of Addition

The addition operation is named +, an overloaded operator of the implementation language. It is a binary infix operator which associates to the left and has a precedence binding of 6. It wil be redefined to work with formulas. In order to accomplish that and still be able to use

the predefined operator, the Prelude[1] should be imported hiding it and it can be used with the notation `(Prelude.+)`.

```
import Prelude hiding ((+))
```

The redefiniton of the operator + for formulas is shown in listing 4.4, where `add` is an auxiliary function that computes the addition.

Listing 4.4: The multiplication operator for formulas.

```
infixl 6 +
(+) :: (?ctxt :: Context) => Formula -> Formula -> Formula
x + y = add [x,y]
```

The function `add` has as input the list of the formulas to be added and produces their sum as the output. In general the resulting formula may be a sum of terms that cannot be further reduced. For example, the following additions result in irreductible sums:

$$7+2x = 7+2x$$

$$(2+3y)+5y = 2+8y$$

$$\frac{xy}{2}+(\frac{4}{3}+xy+z)+(z-1) = \frac{1}{3}+2z+\frac{3xy}{2}$$

In order to build the resulting formula, first the list of resulting terms is built and then a new formula is constructed from this list. In the last example the input list is

$$\left[\frac{xy}{2}, \frac{4}{3}+xy+z, z-1\right]$$

and the list of resulting terms is

$$\left[\frac{1}{3}, 2z, \frac{3xy}{2}\right]$$

from which the sum

$$\frac{1}{3}+2z+\frac{3xy}{2}$$

is built.

[1] The basic standard library of Haskell.

The implementation code for `add` is shown in listing 4.5.

Listing 4.5: The `add` function.

```
add :: (?ctxt :: Context) => [Formula] → Formula
-- add [x1,...,xn]
--   Add the formulas in the list.
--   (Note: each formula can be any formula, including a sum)
--   Merge the formulas trying to add each one with any of the
--   others, to obtain a list of terms that cannot be added
--   together; then make a sum formula out of these terms.
add = mkSum . foldr mergeSum []
```

Function `add` defined using **`foldr`**, a function from the standard library of the implementation language that may be informaly defined as

```
foldr f e [x1,x2,...,xn] =f x1 (f x2 (... (f xn e) ...))
```

Basically **`foldr`** `f e xs` folds the binary operation `f` into the list `xs` with right association, and `e` is the rightmost argument. As an example consider the expression

```
foldr (+) 0 [1,10,48,5]
```

that finds the sum of the elements of the given list of integers. It is equivalent to

```
1 +(10 +(48 +(5 +0)))
```

and is evaluated to `64`. Listing 4.6 shows one possible definition for **`foldr`**. The first argument

Listing 4.6: The `foldr` function.

```
foldr           :: (a → b → b) → b → [a] → b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

of **`foldr`** is a binary function that takes a value of type `a` and a value of type `b` and delivers a value of type `b`. The second argument of **`foldr`** is a value of type `b`. The third argument is of type `[a]`, and the result is a value of type `b`.

The list of resulting terms is built by evaluating the expression

```
foldr mergeSum [] xs
```

(from the definition of `add`). If `xs =[x0,x1,...,xn]`, it is equivalent to

```
mergeSum x0 (mergeSum x1 (... (mergeSum xn []) ...))
```

The function `mergeSum` takes care of combining a value (from the input list) with the values of the (partialy built) result list and is discussed next. The function `mkSum` constructs a new formula that is the sum of the elements found in its argument list, as discussed in section 2.7.5.

As an example of addition consider the reduction steps[2] in the evaluation of

$$(5+x+y)+2x$$

```
  (5+x+y) + 2x
= add [5+x+y, 2x]
= mkSum (foldr mergeSum [] [5+x+y,2x])
= mkSum (mergeSum (5+x+y) (mergeSum (2x) []))
= mkSum (mergeSum (5+x+y) [2x]))
= mkSum [5,3x,y]
= 5+3x+y
```

As another example consider the addition

$$(2+x+3y)+5x+7+(x+2y)+y$$

which can be simplified according to the following reduction steps:

```
  (2 + x + 3y) + 5x + 7 + (x + 2y) + y
= add [2+x+3y, 5x, 7, x+2y,y]
= mkSum (foldr mergeSum [2+x+3y,5x,7,x+2y,y])
= mkSum
    (mergeSum
      (2+x+3y)
```

[2] In these reduction steps the formulas were writen in mathematic notation, not in the implementation language.

```
      (mergeSum
        (5x)
        (mergeSum 7 (mergeSum (x+2y) (mergeSum y [])))))
= mkSum
    (mergeSum
      (2+x+3y)
      (mergeSum (5x) (mergeSum 7 (mergeSum (x+2y) [y]))))
= mkSum
    (mergeSum (2+x+3y) (mergeSum (5x) (mergeSum 7 [x,3y])))
= mkSum (mergeSum (2+x+3y) (mergeSum (5x) [7,x,3y]))
= mkSum (mergeSum (2+x+3y) [7,6x,3y])
= mkSum [9,7x,6y]
= 9+7x+6y
```

`mergeSum` works with two arguments. Its first argument x is a term from the addition that is being carried out. The second argument *ys* is a list of the terms of the sum that will be the result of the addition, which has been partially built so far. `mergeSum` checks if x is a sum or not. If x is a sum $x_1 + x_2 + \ldots + x_n$ then it should be split in its constituent terms x_1, x_2, ..., x_n and each term should be dealt separately. Otherwise it is dealt as a single term x. Each term taken from x in this way is then merged with the terms in *ys*:

- if *ys* is the empty list then the result is the list of terms from x, that is, $[x]$ if x is not a sum or $[x_1, x_2, \ldots, x_n]$ if x is the sum $x_1 + x_2 + \ldots + x_n$.
- if *ys* is not empty then the result is obtained by combining each term from x, that is, x itself if it is not a sum, or x_1, x_2, ..., x_n if x is the sum $x_1 + x_2 + \ldots + x_n$, with the list *ys* using the function `mergeTerm` that is discussed in the sequence. This is accomplished by evaluating the expression

  ```
  mergeTerm x ys
  ```

  in the first case, and

```
foldr mergeTerm ys [x1,x2,...,xn]
```

in the second case.

Listing 4.7 shows the implementation of `mergeSum`.

Listing 4.7: The `mergeSum` function.

```
mergeSum :: (?ctxt :: Context) => Formula -> [Formula] -> [Formula]
-- mergeSum x ys
--   ys is a list [y1, y2, ..., yn] of formulas whose sum
--      y1 + y2 + ... + yn
--   cannot be simplified in the current context.
--   xs is a formula, possibly a sum, to be added to the formula
--      y1 + y2 + ... + yn
--   whose terms are the elements of ys.
--   Basicaly there is a check whether x is a sum. In this case
--   its terms should be merged with ys, one at a time.
--   Otherwise x should be merged directly with ys.
mergeSum x []
    | isSum x   = appArgs x
    | otherwise = [x]
mergeSum x ys
    | isSum x   = foldr mergeTerm ys (appArgs x)
    | otherwise = mergeTerm x ys
```

As examples of how `mergeSum` simplifies consider the following reductions:

```
  mergeSum (5x) []
= [5x]


  mergeSum (2+x+3y) []
= [2,x,3y]


  mergeSum (5x) [7,x,3y]
= mergeTerm (5x) [7,x,3y]
= [7,6x,3y]


  mergeSum (2+x+3y) [7,x,3y]
= foldr mergeTerm [7,x,3y] [2,x,3y]
```

```
= mergeTerm 2 (mergeTerm x (mergeTerm (3y) [7,x,3y]))
= mergeTerm 2 (mergeTerm x [7,x,6y])
= mergeTerm 2 [7,2x,6y]
= [9,2x,6y]
```

`mergeTerm` is applied to two arguments. The first argument x is a formula (but it is not a sum) that must be added to the sum of the formulas that appear as the second argument $ys = [y_1, y_2, \ldots, y_n]$, a list of terms. So the expression `mergeTerm (2x) [5,8x,3y]` finds the sum `(2x) +(5+8x+3y)`. The algorithm for `mergeTerm` has its fundaments in the comutative and associative laws of addition. It has three phases:

1. An attempt is made to add x to one of the the terms y_i in ys. Each term y_i in ys is tried out starting with $i = 1$ until the x and y_i can be successfuly added using `addTerms`. In this case the result is given by combining the sum $x + y_i$ computed by `addTerms` with the list of terms that is obtained by dropping y_i from ys, that is, $[y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n]$ using `mergeSum`. If there is no y_i such that x and y_i could be added using `addTerms` then the second phase starts.

2. Two terms x and y with a common literal part can be added by adding their coefficients. So if $x = c_x l_x$ and $y = c_y l_y$ where c_x and l_x are respectively the coefficient and literal part of x and c_y and l_y are respectively the coefficient and literal part of y and $l_x = l_y = l$ then the sum can be computed as

$$x + y = c_x l_x + c_y l_y = (c_x + c_y) l$$

   In this phase a term y_i from the list ys with the same literal part l of x is searched for by starting with $i = 1$. If found the result is obtained by replacing y_i with the sum of x and y_i computed as explained above in the list ys, if the sum is not equal to 0, in which case y_i is simply removed from ys. If there is no y_i in ys with the same literal part as x then starts the third phase.

3. In this phase x will be inserted into ys to build the result, as there is no way to merge it

with the terms in *ys*. The insertion is carried out using the criteria of maintaining the result list ordered according to the predicate `sorted`. This criteria keeps the simplest terms at the beginning of the list.

The implementation code for `mergeTerm` is shown in listing 4.8

Listing 4.8: The `mergeTerm` function.

```
mergeTerm :: (?ctxt :: Context) => Formula → [Formula] → [Formula]
-- mergeTerm x ys
--   Merge using addition rules the formula x, not a sum, with the
--   terms ys (already merged among themselves), resulting in the list
--   of simplified terms.
mergeTerm x ys = mergeWithTerms ys []
    where
    -- Try to merge the term x and one of the terms of the list
    -- ys (the first that succeeds) using addTerms
    mergeWithTerms (y:ys1) ys2 =
        case addTerms x y of
          Just z  → mergeSum z (reverse ys2 ++ ys1)
          Nothing → mergeWithTerms ys1 (y:ys2)
    mergeWithTerms [] _ =
        mergeLiterals ys []
    -- Try to merge the term x with the terms of the list ys using a broader
    -- rule that checks the literal part of terms and add their coefficients
    -- when the literal parts are the same
    mergeLiterals (y:ys1) ys2
        | litX == litY =
            reverse ys2 ++
            case addTerms coefX coefY of
                Just c  → if isZero c
                             then ys1
                             else mkPro [c,litX] : ys1
                Nothing → error "bug in mergeLiterals"
        | sorted litX litY =
            reverse ys2 ++ x:y:ys1
        | otherwise =
            mergeLiterals ys1 (y:ys2)
        where
        (coefY, litY) = coefLit y
    mergeLiterals [] ys =
        reverse (x:ys)
    (coefX, litX) = coefLit x
```

Note that in the implementation the second and third phase are merged such that only one scan of the list of terms is carried out.

See the examples of applying `mergeTerm` below.

```
  mergeTerm (7xy) []
= mergeWithTerms [] []
= mergeLiterals [] []
= reverse [7xy]
= 7xy

  mergeTerm 5 [6,4x,8y]
= mergeWithTerms [6,4x,8y] []
= mergeSum 9 [4x,8y]
= [9,4x,8y]

  mergeTerm (3x) [6,4x,8y]
= mergeWithTerms [6,4x,8y] []
= mergeWithTerms [4x,8y] [6]
= mergeWithTerms [8y] [4x,6]
= mergeWithTerms [] [8y,4x,6]
= mergeLiterals [6,4x,8y]
= mergeLiterals [4x,8y] [6]
= reverse [6] ++ (3+4)x:[8y]
= [6,7x,8y]

  mergeSum z [6,4x,8y]
= mergeWithTerms [6,4x,8y] []
= mergeWithTerms [4x,8y] [6]
= mergeWithTerms [8y] [4x,6]
= mergeWithTerms [] [8y,4x,6]
= mergeLiterals [6,4x,8y]
= mergeLiterals [4x,8y] [6]
```

```
= mergeLiterals [8y] [4x,6]
= mergeLiterals [] [8y,4x,6]
= reverse (z:[8y,4x,6])
= [6,4x,8y,z]
```

Chapter 5

Multiplication and Subtraction

In this chapter, the author presents the algorithms employed to find the product and the difference of formulas. The framework of the multiplication algorithms follows the general guidelines presented for addition. However, specific multiplication rules replaces those for addition. Therefore, one is advised to brush up on his/her knowledge of addition before reading this chapter.

5.1 Basic Multiplication Rules

Like addition, the multiplication of two arbitrary formulas may be reductible or not, depending on the formulas being multiplied. When it is reductible, the resulting product is a formula computed applying known rules to the factors of the multiplication. For example, the product of two rational numbers is a rational number given by the rules for multiplying rational numbers, e.g.

$$\frac{3}{4}\times\frac{5}{6}=\frac{3\times 5}{4\times 6}=\frac{12}{30}=\frac{2}{5}$$

But the multiplication algorithm may fail in producing a product that is simpler than the stated multiplication. In this case there is no option unless to pack the factors being multiplied in an application formula for products. For example, when multiplying an integer and a literal, e.g.

$$5\times a$$

there is no way to simplify the resulting product and the result is left as an indication of the desired multiplication. Sections 2.1 and 2.7.5 discuss the representation of unreductible formulas, like the one above.

The algorithms used to implement multiplication makes use of a function that tries to multiply two factors producing a reduced formula when possible, or an indication of failure otherwise. This function handles the cases of multiplication by zero, by one, and multiplication of integers, and makes proviso for the handling of special cases when necessary.

5.2 Multiplication of Integers

The most basic rules deal with multiplication by zero, by one, and multiplication of integer numbers.

Zero is the *zero* value of the multiplication operation, that is

$$0 \times x = x \times 0 = 0$$

for any formula x.

One is the *unit* value of the multiplication operation, so

$$1 \times x = x \times 1 = x$$

for any formula x.

Integer multiplication is carried out following rules from elementary school Arithmetics. The implementation makes use of the function

```
(*) :: Integer →Integer →Integer
```

which is taken from the standard library.

Listing 5.1 shows the implementation code for the basic rules discussed in the above sections.

Listing 5.1: The `mulFactors` function.

```
mulFactors :: (?ctxt :: Context) => Formula → Formula → Maybe Formula
-- Try to multiply two factors using simple multiplication rules:
-- 1) multiplication by 0: at least one of the factors is zero
-- 2) multiplication by 1: at least one of the factors is one
-- 3) integer multiplication: both factors are integers
-- 4) proviso for specific rules for the specific kind of involved
--   factors (via a table)
mulFactors x y
    | isInt x             = mulInt1
    | isInt y             = mulInt2
    | isApp x && isJust u = u
    | isApp y && isJust v = v
    | isInd x || isInd y   = Just (mkInd (mkPro [x,y]))
    | otherwise           = Nothing
    where
    u = proT (appFn x) ?ctxt (argList x) y
    v = proT (appFn y) ?ctxt (argList y) x
    mulInt1
        | isInt y             = Just (mkInt (intVal x Prelude.* intVal y))
        | isZero x            = Just zero
        | isOne x             = Just y
        | isApp y && isJust v = v
        | otherwise           = Nothing
    mulInt2
        | isZero y            = Just zero
        | isOne y             = Just x
        | isApp x && isJust u = u
        | otherwise           = Nothing
```

5.3 Handling of Special Cases

Besides the basic rules for multiplication, it is possible to specify special rules. For example, there is no provision for the multiplication of two logarithms in the core multiplication functions. How is it supposed to be achieved? The library offers a mechanism to attach special rules for multiplication into the algorithm that multiplies two formulas. The rules are triggered when none of the explicit rules shown above is adequate to handle the multiplication. For this scheme to work, at least one of the formulas being muliplied should be an application formula.

The core multiplication algorithm has a hook for calling customized functions when none of the basic rules apply. The hook is an overloaded function called `proT`, introduced by the class `ProC` defined in listing 5.2. `proT` is a function with three explicit arguments:

Listing 5.2: An overloaded function for special multiplication rules.

```
class ProC a where
    proT :: (?ctxt :: Context) => a → [Formula] → Formula → Maybe Formula
    proT _ _ _ _ = Nothing

instance (ProC a, ProC b) => ProC (Either a b) where
    proT (Left x)  = proT x
    proT (Right x) = proT x
```

1. the operator of the application formula (the formula corresponding to the first formula to be multiplied),

2. the arguments of this application formula,

3. the second formula to be multiplied.

The function may fail or succeed. If it succeeds the result is a formula corresponding to the product of the two argument formulas. This listing also instantiates the `Either a b` data type to the class. This is important, as the type of the operator in application formulas is an extensible union type, as discussed in sections 2.3 and 2.4. Applying `proT` to a value of the union type is as simple as to apply the appropriate `proT` function to the summand of the union.

The type of each possible operator of an application formula should be an instance of the class `ProcC`, overloading the function `proT` to work with it according to its own rules. Let be the case of multiplying a logarithm and another formula following the rule

$$a \times \log_b x = \log_b (x^a)$$

Listing 5.3 implements this rule by making `Log`, the type of the operator for logarithm formulas, an instance of `ProC`.

Listing 5.3: Multiplication of logarithms.

```
instance ProC Log where
    proT Log ctxt [x,b] a =
        Just (mkLog (x ^ a) b)
```

Not all special rules are covered in this dissertation.

5.4 The Main Part of the Multiplication Algorithm

The multiplication operation is named `*`, which is an overloaded operator of the implementation language. It is a binary infix operator with left association and a precedence binding of 7.

The redefiniton of the operator `*` for formulas is shown in listing 5.4, where `mul` is an auxiliary function that computes the product.

Listing 5.4: The multiplication operator for formulas.

```
infixl 7 *

(*) :: (?ctxt :: Context) => Formula -> Formula -> Formula
x * y = mul [x,y]
```

The function `mul` has as input the list of the formulas to be multiplied and produces their product as the output. In general the resulting value may be a product of factors that cannot be further reduced. For example, the following multiplications result in irreductible products:

$$7 \times y = 7 \times y$$

$$(2 \times a) \times (5 \times a) = 10 \times a^2$$

$$2 \times (a \times b) \times (4 \times a \times c) = 8 \times a^2 \times b \times c$$

In order to build the resulting formula, first the list of resulting factors is built and then a new formula is constructed from this list. In the last example the input list is

$$[2, a \times b, 4 \times a \times c]$$

and the list of resulting factors is

$$[8, a^2, b, c]$$

from which the product

$$8 \times a^2 \times b \times c$$

is built.

The implementation code for `mul` is shown in listing 5.5.

Listing 5.5: The `mul` function.

```
mul :: (?ctxt :: Context) => [Formula] → Formula
-- mul [x1,...,xn]
--   (Note: each formula can be of any form, including a product.)
--   Multiplies the formulas in the list reducing it to the shortest
--   list of formulas that cannot be further simplified under
--   multiplication and then constructs a product from the new list.
mul = mkPro . foldr mergePro []
```

The list of resulting factors is built by evaluating the expression

```
foldr mergePro [] xs
```

(from the definition of `mul`). If `xs =[x0,x1,...,xn]`, it is equivalent to

```
mergePro x0 (mergePro x1 (... (mergePro xn []) ...))
```

The function `mergePro` takes care of combining a formula (from the input list) with the formulas of the (partialy built) result list and is analyzed in the sequence. The function `mkPro` constructs a new formula that is the product of the elements found in its argument list, as analyzed in section 2.7.5.

As an example of multiplication consider the reduction steps of the evaluation of

$$(5xy) \times (2x)$$

```
  mul [5xy,2x]
= mkPro (foldr mergePro [5xy,2x])
= mkPro (mergePro (5xy) (mergePro (2x) []))
= mkPro (mergePro (5xy) [2,x]))
= mkPro [10,x^2,y]
= 10 x^2 y
```

As another example consider the multiplication

$$(2xy^3) \times (5x) \times 7 \times (3xy) \times y$$

which can be simplified according to the following reduction steps:

```
  mul [2 x y^3, 5 x, 7, 3 x y, y]
= mkPro (foldr mergePro [2 x y^3, 5 x, 7, 3 x y, y])
= mkPro
    (mergePro
      (2 x y^3)
      (mergePro
        (5 x)
        (mergePro 7 (mergePro (3 x y) (mergePro y [])))))
= mkPro
    (mergePro
      (2 x y^3)
      (mergePro (5 x) (mergePro 7 (mergePro (3 x y) [y]))))
= mkPro
    (mergePro
      (2 x y^3)
      (mergePro (5 x) (mergePro 7 [3,x,y^2])))
= mkPro (mergePro (2 x y^3) (mergePro (5 x) [21,x,y^2]))
= mkPro (mergePro (2 x y^3) [105,x^2,y^2])
= mkPro [210,x^3,y^5]
= 210 x^3 y^5
```

`mergePro` works with two arguments. Its first argument x is a factor from the multiplication that is being carried out. The second argument *ys* is a list of the factors of the product that will be the result of the multiplication and that has so far been partialy built. `mergePro` checks if x is a product or not. If x is a product $x_1 \times x_2 \times \ldots \times x_n$ then it should be split in its constituent factors $x_1, x_2, \ldots, x_n$ and each factor should be treated separately. Otherwise, it is treated as a single factor x. Each factor taken from x in this way is then merged with the factors in *ys*:

- if *ys* is the empty list then the result is the list of factors from x, that is, $[x]$ if x is not a

product or $[x_1, x_2, \ldots, x_n]$ if x is the product $x_1 \times x_2 \times \ldots \times x_n$.

- if *ys* is not empty then the result is obtained by combining each factor from x, that is, x itself if it is not a product, or $x_1, x_2, \ldots, x_n$ if x is the product $x_1 \times x_2 \times \ldots \times x_n$, with the list *ys* using the function `mergeFactor` that is analyzed in the sequence. This is accomplished by evaluating the expression

  ```
  mergeFactor x ys
  ```

  in the first case, and

  ```
  foldr mergeFactor ys [x1,x2,...,xn]
  ```

  in the second case.

The implementation of `mergePro` follows in listing 5.6.

Listing 5.6: The `mergePro` function.

```
mergePro :: (?ctxt :: Context) => Formula → [Formula] → [Formula]
-- mergePro x ys
--   ys is a list [y1, y2, ..., yn] of formulas whose product
--      y1 * y2 * ... * yn
--   cannot be simplified in the current context.
--   xs is a formula, possibly a product, to be multiplied
--   with the formula
--      y1 * y2 * ... * yn
--   whose factors are the elements of ys.
--   Basicaly there is a check whether x is a product. In this
--   case its factors should be merged with ys, one at a time.
--   Otherwise x should be merged directly with ys.
mergePro x []
    | isPro x   = appArgs x
    | otherwise = [x]
mergePro x ys
    | isPro x   = foldr mergeFactor ys (appArgs x)
    | otherwise = mergeFactor x ys
```

As examples of how `mergePro` simplifies expressions, consider the following reductions:

```
  mergePro (2+x) []
= [2+x]
```

```
  mergePro (2 x y^2) []
= [2,x,y^2]

  mergePro 2 [7,5x,3y]
= mergeFactor 2 [7,5x,3y]
= [14,5x,3y]

  mergePro (2 x y^3) [7,x,y]
= foldr mergeFactor [7,x,y] [2,x,y^3]
= mergeFactor 2 (mergeFactor x (mergeFactor (y^3) [7,x,y]))
= mergeFactor 2 (mergeFactor x [7,x,y^4])
= mergeFactor 2 [7,x^2,y^4]
= [14,x^2,y^4]
```

`mergeFactor` is applied to two arguments. The first argument x is a value (but it is not a product) that must be multiplied to the product of the values that appear as the second argument $ys = [y_1, y_2, \ldots, y_n]$, a list of factors. So the expression `mergeFactor (x^2) [5,8x,3y]` finds the product `(x^2) * (5*8x*3y)`. The fundaments of `mergeFactor` are the comutative and associative laws of multiplication. It works in three phases:

1. An attempt is made to multiply x to one of the the factors y_i in ys. Each factor y_i in ys is tried starting with $i = 1$ until x and y_i can be successfully multiplied using `multiplyFactors`. In this case the result is given by combining the product $x \times y_i$ computed by `multiplyFactors` with the list of factors that is obtained by dropping y_i from ys, that is, $[y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n]$ using `mergePro`. If there is no y_i such that x and y_i could be multiplied using `multiplyFactors` and x is not a reciprocal then the second phase starts. If x is a reciprocal the result is obtained by inserting x as the first or second element in ys, depending on the head of ys being an integer or not, respectively.

2. Two factors x and y with a common base can be multiplied by adding their exponents. So

if $x = {b_x}^{e_x}$ and $y = {b_y}^{e_y}$ where b_x and e_x are respectively the base and the exponent of x and b_y and e_y are respectively the base and the exponent of y and $b_x = b_y = b$ then the product can be computed as

$$x \times y = {b_x}^{e_x} \times {b_y}^{e_y} = b^{e_x + e_y}$$

In this phase a factor y_i from the list ys with the same base b of x is searched for starting with $i = 1$. If found the result is obtained by replacing y_i with the product of x and y_i computed as explained above in the list ys. If there is no y_i in ys with the same base as x then starts the third phase.

3. In this phase x will be inserted into ys to build the result, as there is no way of combining it with the factors in ys. The insertion is carried out using the criteria of maintaining the result list ordered according to the precedence of formulas (section 2.8) and keeping numbers at the beginning of the list. This criteria keeps the simplest factors at the beginning of the list.

Listing 5.7 presents the implementation code for `mergeFactor`.

Note that in the implementation the second and third phases are merged such that, only one scan of the list of factors is completed.

See the examples of applying `mergeFactor` in the sequence.

```
  mergeFactor (a+b) []
= mergeWithFactors [] []
= mergeBases [] []
= reverse [a+b]
= a+b


  mergeFactor 5 [6,x,y]
= mergeWithFactors [6,x,y] []
= mergePro 30 [x,y]
```

```
= [30,x,y]

  mergeFactor (x^2) [6,x^3,y+1]
= mergeWithFactors [6,x^3,y+1] []
= mergeWithFactors [x^3,y+1] [6]
= mergeWithFactors [y+1] [x^3,6]
= mergeWithFactors [] [y+1,x^3,6]
= mergeBases [6,x^3,y+1] []
= mergeBases [x^3,y+1] [6]
= reverse [6] ++ x^(2+3):[y+1]
= [6,x^5,y+1]

  mergePro z [6,x,y]
= mergeWithFactors [6,x,y] []
= mergeWithFactors [x,y] [6]
= mergeWithFactors [y] [x,6]
= mergeWithFactors [] [y,x,6]
= mergeBases [6,x,y]
= mergeBases [x,y] [6]
= mergeBases [y] [x,6]
= mergeBases [] [y,x,6]
= reverse (z:[y,x,6])
= [6,x,y,z]

  mergePro (5^(-1)) [2,x]
= mergeWithFactors [2,x] []
= mergeWithFactors [x] [2]
= mergeWithFactors [] [x,2]
```

```
= [2,5^(-1),x]
```

5.5 Additive Inverse and Subtraction

The additive inverse of a number x is defined as the product of -1 and x. Our implementation language provides an overloaded identifier named **`negate`** for the additive inverse functions. The library redefines it to invert formulas, as shown in listing 5.8.

The subtraction operation is named `-`, and overloaded infix operator of the implementation language. This operator is non associative and has precedence 6. It is also redefine to calculate the difference of two formulas. Listing 5.9 shows its new definition, based on the rule

$$x - y = x \times (-y)$$

Listing 5.7: The `mergeFactor` function.

```
mergeFactor :: (?ctxt :: Context) => Formula → [Formula] → [Formula]
-- mergeFactor x ys
--   Merge, using multiplication rules, the formula x, not a product,
--   with the factors ys (already merged among themselves), resulting in
--   the list of simplified factors.
mergeFactor x ys = mergeWithFactors ys []
    where
    mergeWithFactors (y:ys1) ys2 =
        case mulFactors x y of
          Just z  → mergePro z (reverse ys2 ++ ys1)
          Nothing → mergeWithFactors ys1 (y:ys2)
    mergeWithFactors [] _
        | isInt x        = x:ys
        | isReciprocal x = case ys of
                             y:ys' → if isInt y
                                     then y:x:ys'
                                     else x:ys
                             []    → error "bug in mergeWithFactors"
        | otherwise      = mergeBases ys []
    mergeBases (y:ys1) ys2
        | bX == bY =
            if isRat eX && isRat eY
               then mergePro
                      (bX ^ (case addTerms eX eY of
                               Just e  → e
                               Nothing → error "bug in mergeBases"))
                      (reverse ys2 ++ ys1)
               else reverse ys2 ++ (if sorted eX eY
                                    then x:y:ys1
                                    else y:x:ys1)
        | sorted bX bY && not (isInt y) && not (isReciprocal y) =
            reverse ys2 ++ x:y:ys1
        | otherwise =
            mergeBases ys1 (y:ys2)
        where
        bY = base y
        eY = exponent y
    mergeBases [] ys =
        reverse (x:ys)
    bX = base x
    eX = exponent x
```

Listing 5.8: The `negate` function.

```
negate :: (?ctxt :: Context) => Formula → Formula
negate x = mOne * x
```

Listing 5.9: The subtraction operation.

```
infixl 6 -
(-) :: (?ctxt :: Context) ⇒ Formula → Formula → Formula
x - y = x + mOne * y
```

Chapter 6

Exponentiation and Division

6.1 Mathematical Background

The definition of the natural exponentiation function e^x for $x \in \mathbb{R}$ is given by

$$e^x = \int_0^{+\infty} \frac{1}{x} dx$$

The natural exponential function e^z is defined for all complex numbers $z = x + \imath y$ as

$$e^z = e^x(\cos y + \imath \sin y)$$

The generalized exponential function z^a for $z \in \mathbb{C}^*$ and $a \in \mathbb{C}$ is defined as

$$z^a = e^{a \log z}$$

and can also be written as

$$z^a = e^{a \operatorname{Log} z} e^{i2k\pi a}, \quad k = 0, \pm 1, \pm 2, \ldots$$

As the logarithm function is multi-valued, the generalized exponential function is also multi-valued. Each branch of the logarithm function determines a branch for z^a. The principal value

of z^a is defined as

$$e^{a \operatorname{Log} z}$$

In the particular case where a is integer, the value of z^a is unique and can be given by

$$z^a = \begin{cases} 1 & \text{if } a = 0, \\ z \cdot z^{a-1} & \text{if } a > 0, \\ \dfrac{1}{z^{-a}} & \text{if } a < 0. \end{cases}$$

If $a = \frac{p}{q}$ is rational, then z^a has a finite number of values given by

$$z^{\frac{p}{q}} = e^{\frac{p}{q} \operatorname{Log} z} e^{\imath \frac{p}{q} 2k\pi}, \quad k = 0, 1, \ldots, q-1$$

If a is not rational, then z^a has an infinity of distinct values.

6.2 Basic Algorihtms for Powers

The exponentiation operation is named ^, an overloaded right associativite infix operator of the implementation language, with precedence 8. The redefiniton of the operator ^ for formulas is shown in listing 6.2.

The implementation algorithm for the exponentiation functions on formulas is guided by the structure of the arguments — the base and exponent — with simple case analysis. The most basic cases to be considered in the implementation of the exponentiation function are the following:

Zero exponent

$$z^0 = \begin{cases} \bot & \text{if } z = 0 \\ 1 & \text{if } z \neq 0 \\ z^0 & \text{if it is unknown whether } z \neq 0 \end{cases}$$

Note that:

- The value of 0^0 is undefined.
- The value of z^0 is 1 only if z is known to be different to 0.
- When the base z is not a number, it is not possible to check whether it is zero or not. In this case, it is not possible to simplify the formula z^0. Nonetheless, simplification of $z^0 = 1$ may be forced by setting the `zero_exponent` controller to **`True`** in the context in which the operation is performed.

Zero base

$$0^a = \begin{cases} \bot & \text{if } a = 0 \\ \bot & \text{if } a < 0 \\ 0 & \text{if } a > 0 \\ 0^a & \text{if it is unknown whether } a > 0 \end{cases}$$

Note that

- The value of 0^0 is undefined.
- The value of 0^a is undefined for $a < 0$, since it would mean division by zero: $0^a = \frac{1}{0^{-a}} = \frac{1}{0}$.
- The value of 0^a is 0 only if a is known to be a positive number.
- When the exponent a is not a real number, it is not possible to check whether it is positive or not. In this case it is not possible to simplify the formula 0^a. But it is possible to force the simplification $0^a = 1$ by setting the `zero_base` controller to **`True`** in the context in which the operation is performed.

Unit exponent or unit base

$$\begin{aligned} z^1 &= z \\ 1^a &= 1 \end{aligned}$$

These two simplifications are valid for any z and a.

Non zero integer base and exponent

$$x^a = \begin{cases} x & \text{if } a = 1 \\ x \cdot x^{a-1} & \text{if } a > 1 \\ \dfrac{1}{x^{-a}} & \text{if } a < 0 \end{cases}$$

When $a > 0$, the power x^a can be obtained numerically by means of the standard power function of the implementation library. When the exponent is negative, x^a is the reciprocal of x^{-a}.

Integer powers of the imaginary unit

$$\imath^a = \begin{cases} 1 & \text{if } a \bmod 4 = 0 \\ \imath & \text{if } a \bmod 4 = 1 \\ -1 & \text{if } a \bmod 4 = 2 \\ -\imath & \text{if } a \bmod 4 = 3 \end{cases}$$

Here the remainder of the division of the integer exponent a by 4 is checked and the apropriate value for $\imath^a$ is obtained.

The power $e^\imath$

$$e^\imath = \cos 1 + \imath \sin 1$$

This property is applied only when the controller `trig_exp` [1] is a negative multiple of 7 in the context in which the operation is performed, meaning that the power $e^\imath$ should be expressed using trigonometric functions.

Conversion from power to logarithmic

$$z^a = b^{a \log_b z}, \quad b \neq 0$$

[1] The controller `trig_exp` controlls simplifications related to trigonometric functions.

In the implementation the value used for b is the value of the controller `log_base`[2] defined in the context in which the operation is performed. Its default value is the neperian number e.

This property is applied only when the controller `log_exp`[3] is a positive multiple of 7 in the context in which the operation is performed, meaning that the power z^a should be expressed using the logarithmic function. Another condition for applying this property in the implemented algorithm is that a should not be a number and z should not be zero or a negative number.

Other properties of exponentiation

Any other property of exponentiation considered when simplifying x^y is applied indirectly using two overload functions over the operator types, similarly to the technique use for addition 4.3 and multiplication 5.3. There is an overloaded function for the base of the power, and another for the exponent of the power. Listing 6.1 introduces them. `baseT` and `exponentT` are functions

Listing 6.1: Overloaded functions for special exponentiation rules.

```
class BaseC a where
    baseT :: (?ctxt :: Context) => a → [Formula] → Formula → Maybe Formula
    baseT _ _ _ _ = Nothing

instance (BaseC a, BaseC b) => BaseC (Either a b) where
    baseT (Left x)  = baseT x
    baseT (Right x) = baseT x

class ExponentC a where
    exponentT :: (?ctxt :: Context) => a → Formula → [Formula] → Maybe Formula
    exponentT _ _ _ _ = Nothing

instance (ExponentC a, ExponentC b) => ExponentC (Either a b) where
    exponentT (Left x)  = exponentT x
    exponentT (Right x) = exponentT x
```

with three explicit arguments:

1. the operator of the application formula which is the base (for `baseT`) or the exponent (for `exponentT`) of the exponentiation,

[2] The controller `log_base` holds a formula that is used as the common base for the logarithmic function. Its default value is the neperian number e.

[3] The controller `log_exp` controls simplifications related to the logarithmic function.

2. the arguments of this application formula,

3. the second formula, which is the exponent (for `baseT`) or the base (for `exponentT`).

The functions may fail or succeed. If `baseT` succeeds the result is a formula corresponding to the power of the first formula to the second formula. If `exponentT` succeeds the result is a formula corresponding to the power of the second formula to the first formula. This listing also instantiates the **`Either`** `a b` data type to the classes. This is important, as the type of the operator in application formulas is an extensible union type, as discussed in sections 2.3 and 2.4. Applying `baseT` or `exponentT` to a value of the union type is as simple as to apply the appropriate `proT` or `exponentT` function to the summand of the union.

Non simplifiable powers

When it is not possible to simplify a power, the result will be a new formula representing the exponentiation operation, built using the function `mkPow`.

Implementation code

Listings 6.2 – 6.7 shows the code that implements these basic properties of exponentiation.

6.3 Special properties of exponentiation

Product where a factor is a power

This rule will be triggered when simplifying a product and one of the factors is a power.

$$x \cdot y^a$$

Two cases of simplification are handled here. Other cases of products where a factor is a power are not handled at this point.

Listing 6.2: Power of two terms.

```
infixr 8 ^
(^) :: (?ctxt :: Context) ⇒ Formula → Formula → Formula
x ^ y
    | isInt y                    = powAnyInt x (intVal y)
    | isOne x                    = one
    | isZero x                   = powZeroAny y
    | isE x && isI y &&
      negMult (trig_expd ?ctxt) 7 = cos one + i * sin one
    | x /= logBase &&
      not (less x zero) &&
      not (isRat x) &&
      posMult (log_expd ?ctxt) 7  = logBase ^ y * log x logBase
    | isApp x && isJust u        = fromJust u
    | isApp y && isJust v        = fromJust v
    | isInd x || isInd y         = mkInd (mkPow x y)
    | otherwise                  = mkPow x y
    where
    logBase = log_base ?ctxt
    u = baseT (appFn x) ?ctxt (arguments x) y
    v = exponentT (appFn y) ?ctxt x (arguments y)
```

Division of two integers. The division of two integer numbers x and y is handled by building a fraction $\frac{x}{y}$ followed by its simplification. The result may be an integer, a reciprocal or a fraction.

$$x \cdot y^{-1} = x \cdot \frac{1}{y} = \frac{x}{y} = \frac{\frac{x}{m}}{\frac{y}{m}}$$

where

$$m = \gcd(x, y)$$

Listing 6.3: Power of two terms (continued).

```
  powAnyInt x yVal
    | yVal == 1          = x
    | isInt x            = powIntInt (intVal x) yVal
    | yVal == 0 &&
      zero_exponent ?ctxt = one    -- 0^0 already considered in powIntInt
    | isI x              = powI (mod yVal 4)
    | isApp x && isJust u = fromJust u
    | otherwise          = mkPow x (mkInt yVal)
    where
    u = baseT (appFn x) ?ctxt (arguments x) (mkInt yVal)
```

Listing 6.4: Power of two terms (continued).

```
powIntAny xVal y
    | xVal == 1 = one
    | xVal == 0 = powZeroAny y
    | isApp y && isJust v  = fromJust v
    | otherwise = mkPow (mkInt xVal) y
    where
    v = exponentT (appFn y) ?ctxt (mkInt xVal) (arguments y)
```

Listing 6.5: Power of two terms (continued).

```
powZeroAny y
    | less y zero     = mkInd (mkPow zero y)
    | greater y zero  = zero
    | zero_base ?ctxt = zero
    | otherwise       = mkPow zero y
```

Product of two reciprocal numbers. The product of two reciprocal numbers $x = \frac{1}{p}$ and $y = \frac{1}{q}$ is simplified using the property

$$p^{-1} \cdot q^{-1} = \frac{1}{p} \cdot \frac{1}{q} = \frac{1}{p \cdot q} = (p \cdot q)^{-1}$$

and the result is a reciprocal number. Listing 6.8 shows the function `pro_pow`, which implements these rules.

Power of a product

The power is distributed over the product.

$$(x \cdot y)^z = x^z \cdot y^z$$

The function `base_pro`, defined in listing 6.13, implements this rule.

Power of a power

In the case of power of a power, the exponents are multiplied, as can be seen below.

$$(x^y)^z = x^{y \cdot z}$$

Listing 6.6: Power of two terms (continued).

```
powIntInt xVal yVal
    | yVal == 0 = if xVal == 0
                  then mkInd (mkPow zero zero)
                  else one
    | yVal < 0  = if xVal == 0
                  then mkInd (mkPow zero (mkInt yVal))
                  else let k = xVal Prelude.^ (negate yVal)
                       in  if k < 0
                           then mkPro [mOne, mkPow (mkInt (negate k)) mOne]
                           else if k == 1
                                then one
                                else mkPow (mkInt k) mOne
    | otherwise = mkInt (xVal Prelude.^ yVal)
```

Listing 6.7: Power of two terms (continued).

```
-- Integer powers of the imaginary unit
-- i^n, where i is the imaginary unit, and 0 <= n < 4
powI 0 = one
powI 1 = i
powI 2 = mOne
powI 3 = mI
```

This property is applied when z is an integer number or when the controller `branch`[4] is set to **`True`** in the context in which the operation is performed. Figure 6.14 shows the function `base_pow` that implements this rule.

Power where the exponent is a product

This rule handles formulas that are powers with a product as the exponent

$$z^{x \cdot y}$$

Three cases are considered: the base is e, $\imath$ or the power is a radical.

The base is the neperian number e. It may be possible to simplify powers of e to a product involving $\imath$, as shown by the equalities below.

[4]The **branch** controller controls the simplification of formulas that are multi-valued, like the logarithmic function in the complex domain, by choosing an appropriate branch.

Listing 6.8: Product of power.

```
-- x * (y ^ a)
--   (y^n) * (y^a)  = y^(n+a)   when  bas_exp < 0 && bas_exp' y
--   (k^a) * (y^a)  = (k*y)^a   when  exp_bas < 0 && exp_bas' a
--   x     * (y^-1) = x/y

pro_pow :: (?ctxt :: Context) => [Formula] -> Formula -> Maybe Formula
pro_pow [y,a] x
    | isPow x &&
      bas_exp ?ctxt < 0 &&
      bas_exp' y &&
      basePow x == y       = -- y^n * y^a = y^(n+a)
                             Just (y ^ (exponentPow x + a))
    | isPow x &&
      exp_bas ?ctxt < 0 &&
      exp_bas' a &&
      exponentPow x == a   = -- k^a * y^a = (k*y)^a
                             Just ((basePow x * y) ^ a)
    | isMOne a             = divide x y
    | otherwise            = Nothing
```

Listing 6.9: Product of power: division

```
-- x/y, for any x and y

divide x y
    | isInt y               = divByInt x y
    | isSum y &&
      (den_den ?ctxt > 0 ||
       num_den ?ctxt > 0)   = divBySum x (appArgs y)
    | otherwise             = Nothing
```

$$e^{\imath\frac{\pi}{2}k} = \imath^k, \quad k \in \mathbb{Z}$$

$$e^{\imath\frac{\pi}{2}\frac{p}{q}} = e^{\imath\frac{\pi}{2}\frac{p \bmod (4q)}{q}}, \quad p \in \mathbb{Z}, q \in \mathbb{Z}_+^*$$

$$e^{\imath y} = \cos y + \imath \sin y$$

The use of the third property is controlled by the `trig_exp` controller setting in the context in which the simplification is performed. Exponentials are converted to trigonometric functions only when `trig_exp` is a negative multiple of 7.

Listing 6.10: Product of power: division by an integer

```
-- (x/y), where y is integer

divByInt x y
    | isInt x        = divIntByInt (intVal x) (intVal y)  -- x/y
    | isReciprocal x = Just (mkPow (basePow x * y) mOne)  -- (1/u)/y
    | otherwise      = Nothing
    where

divIntByInt p q
    | k == 1    = Nothing
    | d == 1    = Just (mkInt n)
    | n == 1    = Just (mkPow (mkInt d) mOne)
    | otherwise = Just (mkPro [ mkInt n, mkPow (mkInt d) mOne ])
    where
    k = gcd p q
    n = div p k
    d = div q k
```

The base is the imaginary unit $\imath$ or its symmetric $-\imath$. The applicability of the following properties is checked.

$$\imath^{\frac{p}{q}} = e^{\imath\frac{\pi}{2}\frac{p}{q}}, \quad p \in \mathbb{Z}, q \in \mathbb{Z}_+^*$$
$$(-\imath)^{\frac{p}{q}} = e^{\imath 3\frac{\pi}{2}\frac{p}{q}}, \quad p \in \mathbb{Z}, q \in \mathbb{Z}_+^*$$

These properties are used only when the controller `branch` is set to **`True`** in the context in which the simplification is performed.

The base is an integer number and the exponent is a rational number. In this case, we have the power $k^{\frac{p}{q}}, k, p \in \mathbb{Z}, q \in \mathbb{Z}_+^*$ that can be rewritten as the radical $\sqrt[q]{k^p}$. The sign of the radicand k should be checked. If $k > 0$, then the radical is simplified by extracting as many factors out of the base as possible. However, if $k < 0$, the index q is checked whether it is even or not. The simplification is then carried out as

$$\sqrt[q]{k^p} = \begin{cases} \imath^{2\frac{p}{q}} \sqrt[q]{(-x)^p} & \text{if } q \text{ is even} \\ (-1)^p \sqrt[q]{(-x)^p} & \text{if } q \text{ is odd} \end{cases}$$

These properties are implemented in the code shown on listings 6.15 and 6.16, and must be used only when the controller `branch` is set to **`True`** in the context in which the simplification is performed.

6.4 Division

The division operation is named /, an overloaded infix operator of the implementation language. This operator is left associative and has precedence 7.

It is redefined to calculate the quotient of two formulas. Listing 6.17 shows its new definition, based on the rule

$$\frac{x}{y} = x \times y^{-1}$$

Listing 6.11: Product of power: division by a sum

```
-- x / (y1 + y2 + ... + yn)
--
-- a) (x1 * x2 * ... * xm) / y    = (x1/y) * (x2/y) * ... * (xm/y)
-- b) (1/u) / (y1 + y2 + .. + yn) = 1 / (u*y1 + u*y2 + ... + u*yn)
-- b) x / (y1 + y2 + ... + yn)    = 1 / (y1/x + y2/x + ... + yn/x)

divBySum x ys
    | isPro x             = Just (distribFactors (appArgs x) [] [])
    | isDen x             = distribDen (x ^ mOne)
    | num_den ?ctxt > 0 &&
      num_den' x          = Just (mkPow (distrib' (x ^ mOne) ys) mOne)
    | otherwise           = Nothing
    where
    distribFactors [] l1 l2 =
        mkPro (mergeFactor
                  (mkPow (distrib' (mkPro (reverse l1)) ys) mOne)
                  (reverse l2))
    distribFactors (u:us) l1 l2
        | isDen u             = if den_den ?ctxt > 0 && den_den' v
                                then distribFactors us (v:l1) l2
                                else distribFactors us l1 (u:l2)
        | num_den ?ctxt > 0 &&
          num_den' u          = distribFactors us (v:l1) l2
        | otherwise           = distribFactors us l1 (u:l2)
        where
        v = u ^ mOne
    distribDen u
        | den_den ?ctxt > 0 &&
          den_den' u          = Just (mkPow (distrib' u ys) mOne)
        | otherwise           = Nothing
    distrib' x ys = distrib x ys
                    with ?ctxt = ?ctxt { num_num = den_den ?ctxt
                                       , den_num = num_den ?ctxt
                                       }

-- distrib x ys
--   multiply x and each element of ys and then add the results
distrib x ys =
    mkSum (foldr (\y zs → mergeSum (x * y) zs) [] ys)
```

Listing 6.12: Product of power: overloaded functions

```
instance ProC FnArith where
    proT Pow ctxt xs y = pro_pow xs y with ?ctxt = ctxt
    ...
```

Listing 6.13: Prower of a product.

```
-- (x * y) ^ z = x^z * y^z
-- when exp_bas > 0, distributes the exponent over a product
base_pro :: (?ctxt :: Context) => [Formula] -> Formula -> Maybe Formula
base_pro [x,y] z
    | exp_bas ?ctxt > 0 && exp_bas' z = Just (x^z * y^z)
    | otherwise                       = Nothing

instance BaseC FnArith where
    baseT Pro ctxt xs y = base_pro xs y with ?ctxt = ctxt
    ...
```

Listing 6.14: Power of a power.

```
-- (x ^ y) ^ z
base_pow :: (?ctxt :: Context) => [Formula] -> Formula -> Maybe Formula
base_pow [x,y] z
    | isInt z || branch ?ctxt = Just (x ^ (y * z))
    | otherwise               = Nothing

instance BaseC FnArith where
    baseT Pow ctxt xs y = base_pow xs y with ?ctxt = ctxt
    ...
```

Listing 6.15: Radical.

```
radical :: (?ctxt :: Context) => Integer → Integer → Integer → Formula
radical radicand exp index =
    simplify radicand 1 1 firstPrime otherPrimes
    where
    firstPrime : otherPrimes = primes
    simplify rad root counter p ps
        | mod rad p == 0 = if counter == index
                           then simplify x (root Prelude.* p) 1 p ps
                           else simplify x root (counter Prelude.+ 1) p ps
        | x > p          = let p':ps' = ps
                           in  simplify rad root 1 p' ps'
        | y == 1         = mkInt (root Prelude.^ exp)
        | otherwise      = mkInt (root Prelude.^ exp) *
                           mkPow (mkInt y) (mkRat exp index)
        where
        x = div rad p
        y = div radicand (root Prelude.^ index)

primes :: [Integer]
primes = 2 : filter isPrime [3..]

isPrime :: Integer → Bool
isPrime n = verify primes
    where
    verify (p:ps)
        | p Prelude.* p > n    = True
        | mod n p == 0         = False
        | otherwise            = verify ps
```

Listing 6.16: Radical (continuation).

```
-- x ^ (a * b)
exponent_pro :: (?ctxt :: Context) => Formula → [Formula] → Maybe Formula
exponent_pro x [a,b]
    | isE x =
        let k = a * b / pi2 / i
        in  if isInt k
            then Just (i ^ k)
            else if negMult (trig_expd ?ctxt) 7
                 then let r = k * pi2
                      in  if free r i
                          then Just (cos r + i * sin r)
                          else Nothing
                 else if isRat k && (less k zero || greater k four)
                      then let (n,d) = observe "numDenDrac" numDenFrac k
                                   a = mod n (4 Prelude.* d)
                           in  Just (e ^ (i * pi2 * mkInt a / mkInt d))
                      else Nothing
    | isInt x =
        if isInt a && isReciprocal b && branch ?ctxt
        then Just (radicalInt (intVal x) (intVal a) (intVal (basePow b)))
        else Nothing
    | isI x && isInt a && isReciprocal b && branch ?ctxt =
        Just (e ^ (i * pi * a * b / two))
    | isMI x && isInt a && isReciprocal b && branch ?ctxt =
        Just (e ^ (three * i * pi * a * b / two))
    | otherwise = Nothing
    where
    radicalInt xVal p q         -- x ^ (p/q)     p and q integers
        | xVal > 0     = radical xVal p q
        | mod q 2 == 0 = i ^ (two * a * b) * radical (negate xVal) p q
        | otherwise    = mOne ^ b * radical (negate xVal) p q
```

Listing 6.17: The division operation.

```
infixl 7 /

(/) :: (?ctxt :: Context) => Formula → Formula → Formula
x / y = x * y ^ mOne
```

Chapter 7

System Organization

In this chapter the author presents the module organization of the library. He also explains how to extend the system with new modules. Another discussed topic is performance comparison of the library against similar systems implemented with different approaches.

7.1 Module Organization

The library is organized as collection of modules, described in the following.

SubType Introduces the subtype relationship among two types, based on type classes.

FnClass Introduces the class of operators, `FnC`.

FnArithmetic Defines the data type for the basic arithmetic operators and instantiates it to the `FnC` class.

FnLogarithm Defines the data type for the logarithm operator and instantiates it to the `FnC` class.

Fn Defines the data type for all operators, the supertype `Fn`, using union types and the subtyping relationship introduced in module `SugType`.

Formula Defines the data type for algebraic formulas (the type `Formula`), which is based on the `Fn` type.

Context Defines the data type of contexts (the type `Context`) and a suitable context (`iC` for initial context) for use a top level simplifications. Depends on the `Formula` module, as some controller values are formulas.

Basic Defines basic functions over formulas, needed in almost all other modules. This includes

- checking for canonical form
- finding the arguments of an application formula
- dealing with rational numbers (reciprocals and fractions, numerators and denominators)
- inequalities (checking if a formula is lesser than another formula)
- checking if a formula has a negative coefficient
- getting the coefficient and the literal parts of a formula
- getting the base and the exponent of a power

CBasic Defines type classes for overloading functions that implement special cases of basic operations defined in module `Basic`:

- Class `ArgumentsC` has the funciton `argumentsT` which handles special ways of finding the arguments of an application formula.
- Class `NumeratorC` has the funciton `numeratorT` which handles special ways of finding the numerator of a formula.
- Class `DenominatorC` has the funciton `denominatorT` which handles special ways of finding the denominator of a formula.

Eval Defines functions for dynamic evaluation of formulas.

CEval Defines type classes (with the function `evalT`) to handle special cases of dynamic evaluation of formulas.

Ari Defines basic arithmetic operations with formulas, including

- finding the minum of two formulas
- finding the absolute value of a formula
- addition of formulas
- multiplication of formulas
- exponentation of formulas

CAri Defines tupe classes for overloading the functions that implement special cases of operations like additons and multiplication.

- Class `SumC` has the funciton `sumT` which handles special cases of addition.
- Class `ProC` has the funciton `proT` which handles special cases of multiplication.
- Class `BaseC` has the funciton `baseT` which handles special cases of the base of an exponentiation.
- Class `ExponentC` has the funciton `exponentT` which handles special cases of the exponent of an exponentiation.

Logarithm Defines functions over logarithm formulas. The function **`log`** evalutes the logarithm of a formula.

IBasicLog Defines instances of classes `ArgumentsC`, `numeratorC`, and `DenominatorC` for sums, products and powers, to deal with special cases of basic functions defined in module `Basic` over logarithms.

IBasic Defines instances of classes `ArgumentsC`, `numeratorC`, and `DenominatorC` for sums, products and powers, to deal with special cases of basic functions defined in module `Basic`.

IAriLog Defines instances of classes `SumC`, `ProC`, `BaseC` and `ExponentC` for logarithms, to deal with special cases of the basic arithmetic operations (addition, multiplication and exponentiation) when one argument is a logarithm.

IAri Defines all instances of classes `SumC`, `ProC`, `BaseC` and `ExponentC`, to deal with special cases of the basic arithmetic operations (addition, multiplication and exponentiation).

IEvalAri Defines instances of the classes defined in `CEval` to deal with special cases of dynamic evaluation of basic arithmetic formulas (sums, products and powers).

IEvalLog Defines instances of the classes defined in `CEval` to deal with special cases of dynamic evaluation of arithmetic formulas.

IEval Defines instances of the classes defined in `CEval` to deal with all special cases of dynamic evaluation of formulas.

FormulaParser Defines a parser for formulas, based on monadic parser combinators.

ShowFormula Defines a print printer for formulas.

Main Defines an interactive evaluator of algebra formulas, with a read–eval–print loop.

7.2 Extending the Library

In order to extend the library with new kinds of formulas and/or new operations over formulas, the user should provide the new data type and/or function definition, toghether with instances of any relevant classes that deal with special cases.

Consider the extension of the library with derivatives. The following modules should be added to system to complete the support for the derivative operation:

- `FnDer` with a data type declaration for the operator used with derivative formulas.
- `Der` with the definition of the function `der` for finding derivatives of formulas.
- `CDer` with the definition of a type classe `CDer` that overloads a function `derT` to deal with special rules for finding the derivates. For example, the derivative of a sum is the sum of the derivatives, is a special rule.
- `IDerAri` with instances of class `CDer` that defines the overloaded function `derT` for the basic arithmetic formulas (sums, products and powers).
- `IDerLog` with instances of class `CDer` that defines the overloaded function `derT` for logarithm formulas.
- `IDer` with alls instances of class `CDer`

Some modules should be added or modified to implement special rules concerning derivatives for the operations already defined, like addition, multiplication and exponentiation. For example, the sum of a derivative is the derivative of the summands.

- `IAriDer` with instance definitions of the classes defined in module `CAri`, specifying the rules for adding, multiplying and exponentiating derivatives.
- `IAri` should be modified to include the new instances from IAriDer|.
- Similarly, if the other operations have special rules for operating on derivatives, the corresponding classes should be instantiated accordingly.

As can be noted, the extension of the library with a new kind of formula is reflected in a set of new modules, and the modification of few ones. If the actual Haskell implementations had good support for mutually recursive modules, these modules could be reduced in number, making the system organization simpler.

7.3 Comparison to Other Systems

Most successfull Computer Algebra systems available today have been crafted by a large group of dedicated programmers. This means that special attention could be devoted to the issues of performance of the system, finding most bottlenecks as well as ways of solving them. For instance, the Mathematica system is capable to analyse the code to be processed, seeking for common patterns of simplifications and achieve good performance by handling them as very special cases. If a problem does not fall in such cases, its performance might not be so good.

As a consequence, the implementation of those systems has a very high level of complexity. This, added to the low level of abstraction of their implementation language (mostly are written in C or Lisp), greatly difficults their implementation and mantainability.

Also, in most of those computer algebra systems, the implementation language and the language used for develepment in the system are distinct languages. The last one is usually an interpreted language and, as a consequence, programs developed in these systems have lower performance than programs written in compiled languages.

The computer algebra system developed in this work is implemented in and uses as the language for development, the same language — Haskell. The choice of this language as the implementation language contributes for easing the system development and maintanability, due to the higher level of abstraction found in modern functional languages. It is also expected that program development and maintenance in our system be easier than in other existing algebraic system, for the same reason.

To compare the performance of our system to some other successfull Computer Algebra systems, we run the classical benchmark TDeriv (the Takeuchi derivative). Listing[1] 7.1 shows its implementaion in MatLab. It was run using the derivative implementation provided by MatLab. This program took about 14 minutes to run in a Pentium III 300MHz computer.

Listing 7.2 shows its implementaion in Mapple. When run with the call `tDiff('x^24', 'x^16', 'x^8')`, this program took about 14 minutes to complete, in the same Pentium III 300 MHz computer.

[1]The MatLab and Mapple tests were conducted with the help of Antônio Cláudio Pascoareli Veiga, from the Electrical Engeneering Faculty at the Universidade Federal de Uberlândia.

Listing 7.1: TDeriv in MatLab.

```
function saida = tDeriv(x, y, z);
% Bench mark for the Symbolic Module
if coeff(x, 2) <= coeff(y, 2),
       saida= z;
else
       saida= tDeriv(tDeriv(ddt(x), y, z),
                     tDeriv(ddt(y), z, x), tDeriv(ddt(z), x, y));
end
```

Listing 7.2: TDeriv in Mapple.

```
function saida = tDiff(x, y, z);
  s = 'x' ;
  x = eval ('sym(x)') ;
  y = eval('sym(y)') ;
  z = eval('sym(z)') ;

  polx = sym2poly(x)  ;
  valx = polx(1) ;
  expx = size(polx, 2) - 1 ;

  poly = sym2poly(y) ;
  valy = poly(1)  ;
  expy = size(poly, 2) - 1 ;

  if expx <= expy,
      saida = z
  else
     saida = tDiff(  tDiff(diff(x, s), y, z),
                     tDiff(diff(y,s), z, x),
                     tDiff(diff(z, s), x, y)) ;
end
```

Listings 7.3 and 7.4 show the implementaion in Haskell. The program run in 3 minutes and 30 seconds in the same Pentium III 300 MHz machine.

As the numbers show, our system outperforms the other two used in the tests. This validates the assertion that professional compilers for general purpose languages have better performance than script (domain specific) language implementations found in classic systems like MatLab and Mapple.

Listing 7.3: TDeriv in Haskell (part one).

```
-- Benchmark: Takeuchi derivative

module Main where

import Prelude hiding ((+),(-),(*),(/),(^))
import Formula
import Context
import Basic
import Ari hiding ((+),(*),(^))
import ShowFormula

x = mkVar "x"

-- Polinomial for test: x^24 + x^16 + x^8
-- The result should be: 57657600*x^9

main = putStr (showE result with ?ctxt = iC)

result :: (?ctxt :: Context) => Formula
result = dt y

y = x^mkInt 24 + x^mkInt 16 + x^mkInt 8 with ?ctxt=iC

-- Detection of the steady state of the Takeuchi derivative
-- for the tetrahedron
end :: Formula -> Formula -> Bool
end x y = expon x <= expon y

expon x
    | isPow x = exponentPow x
    | isPro x && isPow x' = exponentPow x'
    where
    _:x':_ = argList x

-- The test itself
dt :: (?ctxt :: Context) => Formula -> Formula
dt k
    | end x y   = z
    | otherwise = dt (dt (deriv x + y + z) +
                      (dt (deriv y + z + x)) +
                      (dt (deriv z + x + y)))
    where
    [x,k'] = argList k
    [y,z]  = argList k'
```

Listing 7.4: TDeriv in Haskell (part two).

```
-- A simple function to find a derivative of a basic formula (a variable,
-- an integer, a contant, a sum, a product or a power) in relation to
-- the variable x
deriv :: (?ctxt :: Context) => Formula → Formula
deriv y
    | isVar y    = if varName y == "x" then one else zero
    | isPow y    = let [b,e] = appArgs y
                   in  (e * b ^ (e - one)) * deriv b
    | isPro y    = let [y1,y2] = arguments y
                   in  deriv y1 * y2 + deriv y2 * y1
    | isSum y    = let [y1,y2] = arguments y
                   in  deriv y1 + deriv y2
    | otherwise  = zero

-- Functions for addition, multiplication and exponentiation redefined
-- so that the arguments are not reordered
infixr 8 ^
(^) :: Formula → Formula  → Formula
(^) = mkPow

infixr 8 *
(*) :: Formula → Formula  → Formula
x * y
    | isInt x && isInt y = mkInt (intVal x Prelude.* intVal y)
    | isZero x = zero
    | isZero y = y
    | isOne x = y
    | isOne y = x
    | isInt x && isPro y =
       let y1:ys = argList y
       in  if isInt y1
           then mkPro (mkInt (intVal x Prelude.* intVal y1) : ys)
           else mkPro [x,y]
    | isInt y && isPro x =
       let x1:xs = argList x
       in  if isInt x1
           then mkPro (mkInt (intVal y Prelude.* intVal x1) : xs)
           else mkPro [y,x]
    | otherwise = mkPro [x,y]

infixr 6 +
(+) :: Formula → Formula  → Formula
x + y
    | isZero x = y
    | isZero y = x
    | isInt x && isInt y = mkInt (intVal x Prelude.+ intVal y)
    | otherwise = mkSum [x, y]
```

Chapter 8

Conclusions

8.1 Conclusions

We feel our objective of constructing a library of functions for high level manipulation of algebraic formulas in a pure functional language has been achieved. Although these days the scenario for programming languages is still dominated by the imperative languages, there is nothing that forbides the use of declarative languages for solving *real world* problems. In particular, the functional languages are ready for use by the software industry and our system is just an example of how that can be done. One can benefit from the higher level of these languages for the description of data and algorithms without making sacrifices to the efficiency of the resulting system, as there are good implementations of functional languages. For instance, the implementation of the Haskell language, the language we have been using in this project, is very good, sometimes better than implementations of conventional languages like C.

One could argue the point that functional languages lack good libraries for general use. This is not always the case. Good libraries are being constantly developed by the functional programming community. We have just added our contribution to the community of functional programmers with a new library of Computer Algebra that can be used with no restrictions.

We worked out algorithms for manipulation of algebraic formulas based on earlier works with non pure functional languages like Scheme and Lisp. Again, it was a new contribution since there was no such system that was not based on any kind of change of state. Even the

ones programmed in the (non pure) functional languages are not declarative since they rely on some sort of state changing.

We developed a library of functions for algebraic manipulation in Haskell, a modern functional language, leading to an embedded domain specif language for Computer Algebra. The library provides types for the representation of algebraic formulas, as well as basic functions (core) for algebraic manipulation. The library is also easily extensible. Our approach was the design and language adequacy analisys.

One difficulty we found was how to express some characteristics of the original algorithms without relying on state changing. The solution was to use implicit parameter passing, which provides a kind of dynamic scoped parameters. This way the context (set of controllers telling in which ways the formulas should be transformed) is not explicitly passed in function applications, keeping the intended arity of operators like $+$. If the transformation of the algebraic formula demanded an auxiliary transformation on (possibly part of) the formula with a different set of controllers, the appropriate function is called with the usual explicitly parameters (the formulas to be manipulated) and a new set of controllers, implicitly passed. The prior set of controllers does not have to be changed. When returning from the auxiliary function, the original controllers are still intact. There is really no change in state during computation. If the user needs or wants to prove correctness of the algorithms she/he implemented, referential transparency allows the substitution of equals for equals, making the reasoning much simpler than in the absence of referential transparency. Undoubtedly this is one of the most appealing advantages of functional programming.

Other major difficult that we dealt with was how to achieve a modular, easily extensible system. It should be easy to add new kind for algebraic formulas and/or operations to the system without breaking any working programs. Two solutions were investigated. The one adopted is based on extensible union data types and is based on a mechanism that permits the extension of a data type with new value constructors (a kind of restricted subtyping). Function extensions is based on overloading with type classes. Another solution is to use existentially quantified type variables and type classes to build an object oriented approach. Each kind of formula has its own data type, which is a subtype of a more general type for all formulas, and a

set of specialized functions.

All solutions to these difficulties are based on concepts not incorparated into the Haskell standard yet. The extended Haskell has shown adequate to the implementation of the library, offering a high level of abstraction.

It is interesting to note that the author's proposal is an embeded system for Computer Algebra aplications development. Building it on top of a well-stablished and high-level language make all the facilities of the hosting language available for the programmer, adding new *features* to it. Few implementations of Computer Algebra systems take this approach. Most of them have their own programming language.

The reasearch has started with a prototype of the library in Scheme[1]. It helped to better understand the algorithms. Then entered Haskell to provide a modern functional language with static type checking. Finally Clean has been targeted, as a more efficient functional language with all the properties of modern functional languages. This work describes just the Haskell part of the research.

8.2 Future Work

Our system uses a context that carries information on how to produce a unique simplification of an expression. This may be restrictive and a more general solution would compute all the possible simplications, eliminating the need of the context. It may be desirable that the algorithms be revised or even redesigned to be free of the need for context. This would be a major work.

A major ambitious project is to extend this system to the context of Pure Algebra and treat things like groups and rings. In this direction there is already DoCon, a work being developed by Sergey Mechveliani [20]. "DoCon joins the categorial approach to the mathematical computation expressed via the Haskell type classes, and explicit processing of the domain description terms."

[1] Appendix A has a short review of what have been implemented in Scheme.

Appendix A

Using the System

This system was first implemented in Scheme as a testbed for the algorithms. This implementation is more complete than what is presented in the thesis body. The following topics have been covered by it:

- Basic arithmetic
- Basic algebra
- Arrays
- Matrixes
- Absolute value
- Differentiation
- Logarithms
- Trignometry
- Equation solving
- Ordinary differential equation solving
- Integrals
- Limits

An interactive system where the user is offered a *read-eval-print loop* where algebraic expressions can be entered, evaluated, and then the result can be printed. This system is similar to the one implemented in Haskell and Clean, but it covers more algorithms.

In the sequence there is some examples of interaction with the system.

```
------------------------------------------------------------------------------
  ICAS-0.1
  An Interactive Computer Algebra System
  Copyright (C) 2007
  José Romildo Malaquias <romildo@iceb.ufop.br>
------------------------------------------------------------------------------
  Type an algebraic expression followed by ; and RETURN to evaluate it.
  The function quit finishes the interactive session.
------------------------------------------------------------------------------

[1] 2 + 5 - 3 - 9;
> -5
[2] john = mary;
> #f
[3] age := 34;
> 34
[4] age;
> 34
[5] 5 + 16 - 3;
> 18
[6] - - - 456;
> -456
[7] 4 3 * -5 ;
> -60
```

```
[8] 4/6;

> 2/3

[9] 5/9 + 7/12;

> 41/36

[10] 3^4;

> 81

[11] 6 + 3*2;

> 12

[12] 6/3*2;

> 4

[13] 2^2^3;

> 256

[14] (7+8)/6;

> 5/2

[15] 7+8/6;

> 25/3

[16] 2/27 - 0.366 + 7.23 10^-2;

> -59299/270000

[17] foo := 20 - 7;

> 13

[18] foo + 2;

> 15

[19] foo := -foo;

> -13

[20] foo*(7 - 5);

> -26

[21] 12345679012345679 9;

> 111111111111111111

[23] @(21)^2;
```

```
> 123456790123456789876543209876543 21

[24] abs(26-41);

> 15

[25] den(4/6);

> 3

[26] gcd(-15,25);

> 5

[27] gcd(0,7);

> 7

[28] lcm(-15,25);

> 75

[29] min(-56,-78);

> -78

[30] min(3/5, 2/3);

> 3/5

[31] mod(8,3);

> 2

[32] num(4/6);

> 2

[33] 5/3;

> 5/3

[34] point := 20;

> 20

[35] 5/3;

> 1.66666666666666666

[36] 1/4;

> 0.25

[37] point := -10;

> -10
```

```
[38] 5342900000;
> 5342900000
[39] point := 1 = 2;
> #f
[40] 0.25;
> 1/4
[41] 4^(1/2);
> 2
[42] (-8)^(1/3);
> -2
[43] 6^(1/2);
> 6^(1/2)
[44] (4 i)^(1/2);
> 2 e^(i pi/4)
[45] 24 ^ (1/3);
> 2 3^(1/3)
[46] 24334 ^ (1/3);
> 23 2^(1/3)
[47] e^(15 i pi / 7);
> e^(i pi/7)
[48] 5!;
> 120
[49] 3^5!;
> 179701029991443121041317982950960503973147562753785110 6401
[50] x + (y + z);
> x + y + z
[51] z*(y x);
> x y z
[52] x y + sin(x) - 3 x/2 + 5 - x^2;
```

```
> 5 - 3/2 x + x y - x^2 + sin x

[53] 3 x + 2 x;

> 5 x

[54] x^5 / x^2;

> x^3

[55] i^2;

> -1

[56] i^7;

> -i

[57] foo := x^2 - 3 x + 5;

> 5 - 3 x + x^2

[58] 2 foo + 6 x;

> 10 + 2 x^2

[59] foo := 'foo;

> foo

[60] 2 foo + 6 x;

> 2 foo + 6 x

[61] expand( (3 x y - 2 y + 5)^2 / x );

> 30 y + 9 x y^2 + 4 y^2/x - 20 y/x + 25/x - 12 y^2

[62] expd( 2 + x/(1 + x) );

> (2 + 3 x)/(1 + x)

[63] fctr( 6 x^2 y - 4 x y^2 / z );

> 2 x y*(-2 y + 3 x z)/z

[64] divout( (x^2 - y^2)/(x^2 + 2 x y + y^2), x );

> (x - y)/(x + y)

[65] parfrac( 1 / ((x + 1)(x - 1)), x );

> 1/(-2 + 2 x) - 1/(2 + 2 x)

[66] rationalize( (2 + i)/(3 - 2 i) );

> 4/13 + 7/13 i
```

```
[67] pgcd( x^2 + 2 x + 1, x^2 - 1, x );
> 2 + 2 x
[68] fctr( pgcd( x^2 + 2 x y + y^2, x^2 - y^2, x) );
> 2 y*(x + y)
[69] pquot( x^3 + 3, x^2 + 1, x );
> x
[70] prem( x^3 + 3, x^2 + 1, x );
> 3 - x
[71] numnum := 0;
> 0
[72] 3 x*(1 - x) (1 + x);
> 3 x*(1 + x) (1 - x)
[73] numnum := 2;
> 2
[74] 3 x*(1 - x) (1 + x);
> x*(1 + x) (3 - 3 x)
[75] numnum := 3;
> 3
[76] 3 x*(1 - x) (1 + x);
> 3 (1 + x) (x - x^2)
[77] numnum := 5;
> 5
[78] 3 x*(1 - x) (1 + x);
> 3 x*(1 + x - x*(1 + x))
[79] numnum := 6;
> 6
[80] 3 x*(1 - x) (1 + x);
> (1 + x) (3 x - 3 x^2)
[81] numnum := 10;
```

```
> 10
[82] 3 x*(1 - x) (1 + x);
> x*(3 + 3 x + x*(-3 - 3 x))
[83] numnum := 15;
> 15
[84] 3 x*(1 - x) (1 + x);
> 3 (x - (x^2 + x^3) + x^2)
[85] numnum := 30;
> 30
[86] 3 x*(1 - x) (1 + x);
> 3 x - 3 x^3
[87] denden := 15;
> 15
[88] y/x 1/(1+x) 1/(1-x);
> y/(x - x^3)
[89] dennum := 6;
> 6
[90] (x + 3) / 3 / x;
> 1/3 + 1/x
[91] numden := 0;
> 0
[92] (3 + x)/(1 + x);
> (3 + x)/(1 + x)
[93] numden := 30;
> 30
[94] numden := 5;
> 5
[95] (3 + x)/(1 + x);
> 1/(x/(3 + x) + 1/(3 + x))
```

```
[96] numden := 30;

> 30

[97] (3 + x)/(1 + x);

> 1/(1/(1 + 3/x) + 1/(3 + x))

[98] basexp := 3;

> 3

[99] x^(1+y);

> x^(1 + y)

[100] baseexp := 3;

> 3

[101] x^(1+y);

> x x^y

[102] pwrexpd := 6;

> 6

[103] pwrexpd := 6;

> 6

[104] (1+x)^3 / (1+x+y)^2;

> (1 + 3 x + 3 x^2 + x^3)/(1 + 2 x + 2 y + 2 x y + x^2 + y^2)

[105] (x^2)^(1/2);

> x

[106] pickbranch := 1=2;

> #f

[107] (x^2)^(1/2);

> x

[108] pickbranch? := 1=2;

> #f

[109] (x^2)^(1/2);

> (x^2)^(1/2)

[110] solve(x^2 == 4 a, x);
```

```
{ x==-2 a^(1/2),
  x==2 a^(1/2) }
[111] [3,p] + [2, abs q];
[5, p + abs q]
[112] log(200,10);
2 + log(2, 10)
[113] logexpd(ln(x y^2/z), 15);
ln x + 2 ln y - ln z
[114] sin(11 pi/15);
(7/16 - 5^(1/2)/16 + (30 - 6 5^(1/2))^(1/2)/16)^(1/2)
[115] dif(a x^3, x);
3 a x^2
[116] integral(4 + a x^2, x);
4 x + a x^3/3 + ARB(1)
```

Bibliography

[1] WINKLER, F. *Polynomial algorithms in computer algebra*. Springer-Verlag, 1996.

[2] BUCHBERGER, B., COLLINS, G. E., LOOS, R. (Eds.). *Computer algebra, symbolic and algebraic computation*. 2nd. ed. Springer, 1983.

[3] WHITEHEAD, A. N., RUSSELL, B. *Principia mathematica*. Cambridge University Press, 1925. v. 1, Cap. Appendix 3.

[4] DILLER, A. *Compiling functional languages*. John Willey & Sons Ltda, 1988.

[5] RUSSEL, B. *Logic and knowledge: Essays*. George Allen & Unwin, 1956.

[6] PLASMEIJER, R., VAN EEKELEN, M. The Clean language report, version 2.1. Technical report, University of Nijmegen, 2002.

[7] NILSSON, N. J. *Principles of artificial intelligence*. Morgan Kaufmann, 1980. p. 35–47.

[8] CHANG, C. M. *Mathematical analysis in engineering*. Cambridge University Press, 1994.

[9] HEARN, A. Reduce: A user-oriented interactive system for algebraic simplification. In: KLERER, M., REINFELDS, J. (Eds.) *Interactive Systems for Experimental Applied Mathematics*. : Academic Press, 1968. p. 79–80.

[10] BRACKX, F., CONSTALES, D. *Computer algebra with lisp and reduce*. Kluwer Academic Publishers, 1992.

[11] ANDERSSON, G. *Applied mathematics with maple*. Chartwell-Bratt, 1997.

[12] ARNEY, D. C. *Exploring calculus with derive*. Addison-Wesley Publishing Company, 1992.

[13] BRADLEY, G. L., SMITH, K. J. *Calculus*. Prentice Hall, 1995.

[14] ANDREW, A. D., CAIN, G. L., CRUM, S., MORLEY, T. D. *Calculus projects using mathematica*. McGraw-Hill, 1996.

[15] LIANG, S., HUDAK, P., JONES, M. Monad transformers and modular interpreters. In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Priciples of Programming Languages*. San Francisco, CA: ACM Press, January 1995.

[16] JONES, S. P. Explicit quantification in haskell. http://research.microsoft.com/users/simonpj/Haskell/quantification.html, February 2007.

[17] LEWIS, J., SHIELDS, M., MEIJER, E., LAUNCHBURY, J. Implicit parameters: Dynamic scoping with static types. *27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, 2000.

[18] WADLER, P. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In: *Proc. of 2nd International Conference on Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag, 1985. p. 113–128.

[19] JONES, S. P., MARLOW, S., ELLIOTT, C. Stretching the storage manager: weak pointers and stable names in haskell. *IFL*, 1999.

[20] MECHVELIANI, S. Docon: the algebraic domain constructor. http://haskell.org/docon/, February 2007.

[21] KLERER, M., REINFELDS, J. (Eds.). *Interactive systems for experimental applied mathematics*. New York: Academic Press, 1968.

[22] JONES, M. P., JONES, S. P. Lightweight extensible records for haskell. *Proceedings of the 1999 Haskell Workshop*, October 1999.

[23] NORDLANDER, J. Polymorphic subtyping in o'haskell. In: *Science of Computer Programming*. : Elsevier, 2002. v. 43, p. 93 – 127. Extended version.

[24] DAVIE, A. J. T. Algebraic formula manipulation in a functional language: A first attempt. *Functional Programming*, 1995.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo