Simplification Infrastructure for an Implementation of the Chiron Logic

Simplification Infrastructure for an Implementation of the Chiron Logic

By
Han Yin Zhang, B.S.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF COMPUTER SCIENCE (2010) McMaster University
(Department of Computing and Software) Hamilton, Ontario

TITLE: Simplification Infrastructure for an Implementation of the Chiron Logic

AUTHOR: Han Yin Zhang, B.S. (Nanjing University, Nanjing, China)

SUPERVISOR: Dr. William M. Farmer

NUMBER OF PAGES: ix, 77

Abstract

Simplification is an important and heavily used facility in many mathematical software systems including both computer algebra systems and computer theorem proving systems. The objective of the MathScheme project is to develop a new generation of mechanized mathematic systems that combines the advantages of both computer algebra and computer theorem proving. Serving as the underlying logic of MathScheme, Chiron is used to formalize mathematics in our project. Therefore, we want to build a simplifier that simplifies Chiron expressions for the MathScheme project. This thesis presents the design and implementation of a simplification infrastructure that allows users to build their own simplifiers. This framework can be viewed as a customizable simplifier. It provides a set of simplification strategies and mechanisms for managing contexts. The rules module of this framework allows future developers to define new simplification rules and add them into the rule library. Using different strategies and optional arguments, developers can build simplifiers that work in various ways. The ultimate goal of this framework is to provide a powerful tool with good flexibility so that other people can use it as a handy building block or an experimental environment in the future development and application of MathScheme.

Acknowledgments

I would like first extend my sincere gratitude to my supervisor, Dr. William M. Farmer, for his support, guidance, and lots of help during my graduate studies at McMaster. Without his inspiration and expertise, I would not have been able to complete this thesis.

My special thanks and appreciation goes to the my examination committee members, Dr. Jacques Carette and Dr. Ridha Khedri, for their valuable comments and advice.

Finally, I would like to thank my family, my father Xu Zhang and mother Xuexun Xiao, for their endless love, encouragement, and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Simplification in MathScheme

MathScheme [15], which was originated at and is being carried out by McMaster University, is a project to mechanize mathematics and support a wide range of mathematical activity. The objective of MathScheme is to create a mathematical software system that is both powerful and trustworthy by combining the strengths of computer algebra systems and computer theorem proving systems. To achieve this goal, we need mechanisms and tools to represent mathematical knowledge and automate mathematical reasoning. Presently, some theories and components such as Chiron [8], biform theories [6], Mei [23], an implementation of Chiron in OCaml [19], and a prototype theory library have already been developed or partly developed in previous work.

The information produced by a mechanized mathematics system is often overwhelming to the user because too many details and steps are displayed. Consequently, the user wants to simplify expressions to make them more understandable. Simplification is a process that transforms one expression to another expression. The two

expressions denote the same value, but the syntactic form of the output expression is intended to be "simpler" than that of the input expression. Simplification is usually intertwined with both symbolic computation and formal deduction. It is important and pervasive in mathematics software. However, it is not easy to find a formal definition of simplification because from different points of view the criteria of simplicity are different [2]. In MathScheme, a simplifier is designed to be a tool that can be used directly by end users or other processes of the system. Such a simplifier works on the syntactic forms of Chiron expressions because Chiron is the underlying logic of MathScheme. The ideas of this thesis could be applied to expressions in other logics as well.

As part of the MathScheme project, this thesis presents the implementation of a framework for simplifying Chiron expressions. Furthermore, this framework gives flexibility to developers so that developers can build their own simplification rules and combine different simplification strategies to yield their own simplifier. It is also possible, with the right kind of user interface software, for end users to build their own simplifiers. The source code of my work can be found at

`http://imps.mcmaster.ca/hanyinzhang/`.

1.2 Organization of the Thesis

The implementation of our simplifier framework divides into several modules. Thus, this thesis is organized according to the modules of our system. The following is the outline of the thesis.

Chapter 2 presents the background concepts for our work. A description of MathScheme and Chiron is found in this chapter.

Chapter 3 gives an overview of our problem and the requirements for our framework.

Chapter 4 describes the path module. A path is a concept which assists us in navigating the syntax trees of expressions. Apart from a simplifier, the path module can be used by other routines or functions in our system.

Chapter 5 is about contexts. Contexts play an important role in simplification. Representation and management of a context are the major topics of this section.

Chapter 6 discusses simplification rules. In this chapter, we mainly illustrate the implementation of the built-in rule tables.

Chapter 7 presents the main module of our framework. It discusses strategies for doing simplification, the organization of our system, and the algorithms of some major functions.

Chapter 8 introduces some useful tools in our framework. These tools help people manipulate sub-expressions and local contexts easily. Besides being used by simplifiers, they can be used in future development as well.

Chapter 9 is the conclusion chapter. It gives a summary of my work and proposes some future work to help improve our framework.

1.3 Fonts

Several fonts are used in this thesis for special purposes:

- *Italics* — for designating the term that is being defined in a definition.

- Sans serif — for the names of Chiron symbols such as op-app.
- **Bold** — for the OCaml types that represent sorts of Chiron expressions such as **sexpression**.
- `Typewriter` — for OCaml code such as the function name `replace`.

Chapter 2

Background

2.1 Mechanized Mathematics System

In the introduction of [6], a definition of a *mechanized mathematics system* (MMS) is given. An MMS is a computer software system that provides models to represent mathematics and mathematical activities. The two major branches of MMSs are *computer algebra systems* (CASs) and *computer theorem proving systems* (CTPSs). CASs work well in symbolic computation. The mathematical knowledge is represented procedurally as algorithms in CASs. CTPSs are good at formal deduction. In CTPSs, the mathematical knowledge is represented declaratively as axioms.

When people perform computations, they may find a CAS is convenient to use. However, the computations in a CAS do not represent fully rigorous processes. Therefore, the results of a CAS can be incorrect sometimes. Compared with a CAS, a CTPS is hard to use and often computationally inefficient. The soundness of the reasoning process is the strength of a CTPS. In several ways, the features of CASs and CTPSs are complementary.

2.2 MathScheme

In [13] and [5], a big picture of MathScheme, which is intended to be a new generation of MMS, is presented. The first goal of MathScheme is to combine computer algebra and computer theorem proving in a framework for mechanized mathematics. Building a mathematics knowledge management (MKM) system on top of this framework is the next goal. Our long-range goal is to use the system and its library to produce an interactive mathematics laboratory which has the potential to change the current way of doing mathematics.

The notion of a biform theory is the key idea for realizing the integration of computer algebra and computer theorem proving. In a theory, we use axioms to define different elements and the relationship between them. The operators which do computation in a theory are usually represented by algorithms. In a word, axioms describe mathematical knowledge declaratively while algorithms represent mathematical knowledge procedurally. A biform theory contains both a set of axioms and a set of algorithms. In order to formalize biform theories, we need a logic like Chiron which gives facilities for expressing different facets of mathematics process.

More information about MathScheme project can be found at its homepage [15].

2.3 Chiron

Chiron [7, 8], designed and developed by Dr. William Farmer, is a multi-paradigm, higher-order logic. It is designed to be a facility for mechanizing mathematics. A logic that is intended to serve as the logical basis for an MMS needs to possess both theoretical expressivity and practical expressivity [7]. Traditional logics like von-

2. Background

Neumann-Bernays-Gödel (NBG) set theory and Zermelo-Fraenkel (ZF) set theory are highly expressive in theory but they are not easy to use in practice. Chiron is derived from NBG set theory and supports several of the reasoning paradigms described in [7]. By using Chiron, people can reason about both the syntax and semantics of an expression. Chiron is ideal for formalizing biform theories [6] and is a suitable foundation for a practical, general-purpose MMS.

In Chiron, expressions are symbols or tuples of expressions. They are essentially the same as S-expressions in Lisp. The notation of an expression of Chiron is defined as below:

Expr-1 (Atomic expression)

$$\frac{s \in \mathcal{S}}{\mathbf{expr}[s]}$$

Expr-2 (Compound expression)

$$\frac{\mathbf{expr}[e_1], \ldots, \mathbf{expr}[e_n]}{\mathbf{expr}[(e_1, \ldots, e_n)]}$$

where $n \geq 0$.

In these formation rules, $\mathcal{S}$ is a infinite set of Chiron symbols and $\mathbf{expr}[e]$ asserts that e is an expression. An expression can be proper or improper. Proper expressions denote values while improper expressions are nondenoting. Each expression has the structure of a tree. Symbols, which are improper expressions, serve as the leaves of the tree. A proper expression is a compound expression which consists of other expressions. Conversely, not all compound expressions are proper expressions. In Appendix B of the Chiron technical report [8], 19 formation rules define the notion a proper expression. A category of proper expressions is associated with each formation rule. The corresponding official notation and compact notation of proper expressions

are listed in Table 2.1. In this thesis (as in the Chiron technical report [8]), we will use $s, t, u, v, w, x, y, z, \ldots$ to denote symbols; $o, o', \ldots$ to denote operator names; $O, O', \ldots$ to denote operators; $\alpha, \beta, \gamma, \ldots$ to denote types; $a, b, c, \ldots$ to denote terms; $A, B, C, \ldots$ to denote formulas; and $k, k', \ldots$ to denote kinds.

The compact notation is more readable and convenient to use. We mainly use the compact notation to describe Chiron expressions and examples in the rest of this thesis. The official notation shows the exact structures of expressions. When we want to talk about how an expression is constituted, we will use the official notation instead of the compact notation.

2.4 Chiron Implementation

As a component of MathScheme project, Ni Hong began an implementation of Chiron in Objective Caml (OCaml) [1] as part of his master's research [19]. One of the most important parts of his work is using the host programming language's type system to implement the Chiron expression structure. OCaml was chosen to fulfill this task, in part, because of its strong static typing. The implementation of Chiron in OCaml uses polymorphic variant types to define overlapping algebraic data types that we can use to represent Chiron expressions. An algebraic data type in OCaml is a type whose values are tagged by constructors. A tagged value consists of values from other data types. In the Chiron implementation, a set of algebraic data types are defined to represent Chiron expressions. A value of these expression types can be a compound value which consists the values of other expression types. Moreover, an algebraic data type could be defined as a part of another algebraic data type. Therefore, we can define an expression type as a subtype of another expression type.

Table 2.1: Chiron notations for proper expressions

EXPRESSION SORT	OFFICIAL NOTATION	COMPACT NOTATION
Operator	$(\mathsf{op}, o, k_1, \ldots, k_{n+1})$	$(o :: k_1, \ldots, k_{n+1})$
Operator application	$(\mathsf{op\text{-}app}, O, e_1, \ldots, e_n)$	$O(e_1, \ldots, e_n)$
Constant	(con, o, k)	$[o :: k]$
Variable	$(\mathsf{var}, x, \alpha)$	$(x : \alpha)$
Type application	$(\mathsf{type\text{-}app}, \alpha, a)$	$\alpha(a)$
Dependent function type	$(\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$	$(\Lambda\, x : \alpha\, .\, \beta)$
Function application	$(\mathsf{fun\text{-}app}, f, a)$	$f(a)$
Function abstraction	$(\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$	$(\lambda\, x : \alpha\, .\, b)$
Conditional term	(if, A, b, c)	$\mathsf{if}(A, b, c)$
Existential quantification	$(\mathsf{exists}, (\mathsf{var}, x, \alpha), B)$	$(\exists\, x : \alpha\, .\, B)$
Unique existential	$(\mathsf{uni\text{-}exists}, (\mathsf{var}, x, \alpha), B)$	$(\exists !\, x : \alpha\, .\, B)$
Universal quantification	$(\mathsf{forall}, (\mathsf{var}, x, \alpha), B)$	$(\forall\, x : \alpha\, .\, B)$
Definite description	$(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$	$(\iota\, x : \alpha\, .\, B)$
Indefinite description	$(\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$	$(\epsilon\, x : \alpha\, .\, B)$
Set Construction	$(\mathsf{set\text{-}cons}, a_1, \ldots, a_n)$	$\{a_1, \ldots, a_n\}$
List Construction	$(\mathsf{list\text{-}cons}, a_1, \ldots, a_n)$	$[a_1, \ldots, a_n]$
Class abstraction	$(\mathsf{class\text{-}abs}, (\mathsf{var}, x, \alpha), B)$	$(\mathrm{C}\, x : \alpha\, .\, B)$
Quotation	(quote, e)	$\ulcorner e \urcorner$
Evaluation	(eval, a, k)	$[\![a]\!]_k$

Among several modules in the system, `Types` and `Keywords` establish the fundamental type system of Chiron and offer interfaces for building Chiron expressions. In the `Keywords` module, the key words of Chiron are defined. As we mentioned in the last section, every S-expression corresponds to a tree structure. The polymorphic variant type is ideal for representing the syntax trees of Chiron expressions. In the `Types` module, all Chiron expressions including proper expressions and improper expressions are formalized as S-expressions by using polymorphic variant types. The definitions of the polymorphic variant types are recursive and overlapping. In the Chiron implementation, proper expressions are organized into four OCaml types: **operator**, **ctype**, **term**, and **formula**. Table 2.2 shows the corresponding expressions of four proper expression types.

All these types are subtypes of **proper** and **proper** is a subtype of **sexpression**. For example, a definite description has the form:

$$(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$$

in Chiron, and the form

$$\texttt{DefDes}\ (x, \alpha, B)$$

in the Chiron implementation. `DefDes` is the constructor for a definite description. The definite description contains three components which are the name x of the variable (type **symbol**), the type α of the variable (type **ctype**) and the body (type **formula**). The constructors of variant types roughly correspond to the leading symbols of Chiron proper expressions. Every constructor is applied to a tuple of values that represent the components of the proper expression.

The simplification infrastructure presented in this thesis works on the OCaml implementation of Chiron expressions. It is thus natural to implement my work in

Table 2.2: Implementation of Chiron expressions

Data Type	Constructor	Description
operator	`Operator`	operator
ctype	`TConstant`	type constant
	`TOpApp`	type operator application
	`TTypeApp`	type application
	`TDepFunType`	dependent function type
	`TEval`	type evaluation
term	`Constant`	term constant
	`OpApp`	term operator application
	`Var`	variable
	`FunApp`	function application
	`FunAbs`	function abstraction
	`If`	conditional term
	`DefDes`	definite description
	`IndefDes`	indefinite description
	`Quote`	quotation
	`Eval`	term evaluation
formula	`FConstant`	formula constant
	`FOpApp`	formula operator application
	`FExists`	existential quantification
	`FForall`	universal quantification
	`FEval`	formula evaluation

OCaml. Besides the purpose of following the implementation of Chiron expressions, we can make use of the advantages of OCaml programming language.

2.5 Simplification

In many computer algebra systems and computer theorem provers such as IMPS [9, 10], Isabelle [20], Maple [4], and Mathematica [22], simplification is an important and heavily used routine. Besides mathematics software, simplification is often used in program analysis and program transformation systems as well. A technique for combining decision procedures to do simplification is proposed in [18]. On the concept of simplification, some papers like [2] and [17] have good discussions. The common ideas we find among different simplifiers have helped us design our simplification infrastructure. In the MathScheme framework, a simplifier is a transformer which maps expressions to expressions. The input expression and output expression may have different representations but must denote the same semantic value. In the ideal case, the output expression is simpler than the input expression. Ultimately, we hope to obtain the simplest form of an expression. However, simplification is also controversial because the concept of "simpler" varies from case to case. In [2], a general definition of simplicity is given. An expression A is simpler than an expression B (in a theory T) if the length of the syntactic form of A (as defined in T) is shorter than the length of the syntactic form of B (assuming A and B are semantically equivalent). Some related topics such as the canonical form (or normal form) of an expression are discussed in [3]. Furthermore, simplicity of syntactic form in MMS mainly refers to two aspects of an expression. On the one hand, simplicity means an expression is easy to comprehend by a human, and on the other hand, simplicity means an expression is easy to be manipulated by a computer system. Sometimes these two aspects conflict with each other.

2. Background

The major routine of our simplification is repeated application of simplification rules to a given expression. Some rules that are represented by equations are called rewrite rules. For example, we have a rewrite rule, $x+x+x = 3\times x$. By applying this rule, we can simplify expression $5 + 5 + 5$ to 3×5. Besides rewrite rules, some rules which are designed to deal with complex tasks cannot be represented by equations. Our framework is intended to take different kinds of simplification rules as long as all rules are implemented in a unified way. Details about simplification rules are discussed in chapter 6. The result of simplification also depends on the context or set of background assumptions. In different contexts, the simplifier may produce different results. Contexts are discussed in chapter 5.

Chapter 3

Problem

3.1 Simplifier for Chiron

A simplifier plays an important role in our mechanized mathematics system. It provide facilities which help the end users accomplish many detailed and tedious tasks. For example, we often need to deal with evaluation in MathScheme. In most cases, the evaluation of a quotation can be simplified by using a simple rule. Chiron, the core logic of MathScheme, is used to formalize mathematics in our project. On top of the system that implements Chiron expressions, we want to build a simplifier. It is intended to assist users, both developers and end users, in simplifying Chiron expressions.

A powerful Chiron simplifier needs to be supported by a large number of rules. In the future development of MathScheme, new components such as operators and functions will be defined in Chiron. Consequently, new simplification rules will be built to handle these expressions. Instead of making a fixed simplifier, we want to create a simplification framework which can be extended in the future.

3.2 A Simplification Infrastructure

In this thesis, we will present an infrastructure for building simplifiers. It allows users to build their own simplifier by using different strategies and optional arguments. The ultimate goal of this framework is to provide a powerful tool with good flexibility so that other people can use it as a handy building block or an experimental environment in their future development. The design goals of our framework are listed as follows.

(1) *To provide a mechanism for defining and using simplification rules.*
Users can define their own simplifications rules to create the built-in rule library which will be used by the simplifier. Besides built-in rules, users can also define some temporary rules and store them in a rule list which will be used by the simplifier in some specific modes. The built-in rule library is the default rule source we use in many simplification modes. The user-defined rule list mode is another option we can use to have some flexibility.

(2) *To provide tools for navigating and manipulating sub-expressions.*
A path is an ideal tool that helps us identify a location in an expression. By using paths, it is easy for a user to do operations on sub-expressions and local contexts.

(3) *To provide different strategies for the simplifier to traverse the syntax trees of expressions and apply simplification rules to these expressions.*
Different strategies have their own benefits when they are used to handle different expressions. The simplifier may run through an expression more quickly in some strategy while the same expression can be simplified completely by applying another strategy. The users can choose one strategy or combine several strategies to meet their requirements in simplifications.

(4) *To provide tools for managing and using a context of background assumptions.*

3. Problem

The result of a simplification can be different in different contexts. By taking global and local contexts into account, our system is capable to handling conditional simplification rules. Furthermore, in many cases, taking advantage of local context can speed up the simplification process significantly.

Chapter 4

Paths in Expressions

4.1 Syntax Trees of Chiron Expressions

In chapter 2, we have introduced the structure of Chiron expressions and their implementations. To put it simply, every S-expression corresponds to a syntax tree. In the implementation, the syntax tree of an expression is represented by a value in a polymorphic variant type. In this kind of tree structure, when we talk about a node, we often mean the subtree whose root is that node instead of a single node which represents a symbol.

Our simplifier is intended to work on Chiron expressions. When we try to simplify an expression, we need to analyze its syntax tree. Some special techniques will be applied to deal with the syntax tree of Chiron expressions. The notion of a *path* is one of the most important ideas used in our simplification process.

4.2 Paths in Syntax Trees

A path is a simple and useful concept for navigating in a tree structure. We can go to some specific node by using a path. We can keep track of the traversal process of a tree by recording a path. Paths play an important role in the simplification and calculation of a local context. It is a handy tool which is easy to use and understand. Before we give the definition of path, we need to first number the nodes of a tree. At every level, the nodes are numbered in a order from left to right. The leftmost node at every level has the same number.

Definition 1 *A path is a list of integers which designates a certain node in a tree.*

For example, suppose we have a tree as shown in Figure 4.1. Every number in a

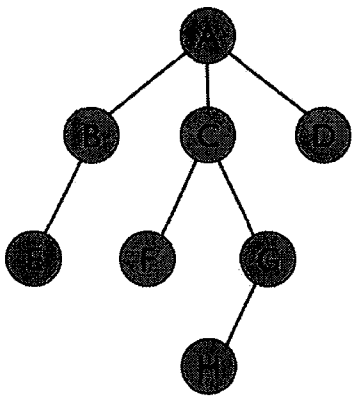

Figure 4.1: Tree example

path represents a node at a certain level. Let the index start from 0. At the top level, 0 represents node A. At the second level, 0, 1 and 2 represent node B, C and D respectively. In that way, path $\langle 0, 1 \rangle$ denotes node C, path $\langle 0, 1, 1, 0 \rangle$ denotes node H. With paths, it is easy to talk about a certain sub-expression in a syntax tree.

Especially in our work, paths are very useful because the data structure of Chiron expression is fixed. We do not have any "place" to attach new information to a node. We can use a path as an index to represent a certain node in a tree. By using a path, we can indirectly link information such as a local context to the corresponding node. Now, the problem is how should we calculate a path in our implementation.

4.3 Sub-expression or Expression Component

There is a small difference between the representation of Chiron expressions in theory and that in implementation. In Chiron, the official notation defines the structure of expressions. The formation rules of proper expressions are formal and recursive. The implementation of Chiron expressions is based on the official notation, but does not represent Chiron expressions literally. Instead the implementation introduces certain optimizations and simplifications, using a concise fashion to represent some expressions. However, sometimes we cannot obtain the sub-expression information directly from a Chiron expression in the implementation. For example, existential quantification in Chiron is written as $(\mathsf{exists}, (\mathsf{var}, x, \alpha), B)$. In our implementation, existential quantification is represented like `FExists` (x, α, B). The syntax trees of these two expressions are shown in Figure 4.2. The root node of the right tree (implementation representation) has three child nodes while the root node of the left tree (official notation) has two child nodes. The two children of the left tree root node are a variable and a formula. They are both proper Chiron expressions. Therefore, acquiring a sub-expression from the left tree is straightforward. In the right tree, the first (leftmost) child of the root node is a symbol x. The symbol x is not a sub-expression of the variable because a symbol is not a proper expression. According to the definition of sub-expression in Chiron technical report, only a proper expression can be a sub-expression. If we extract the first child node from an OCaml expression

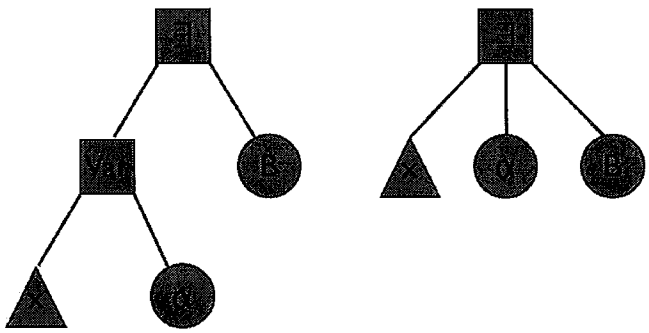

Figure 4.2: Syntax trees of two existential quantifier expressions

of an existential quantifier, we do not acquire a variable. However, the type information of the variable is indeed stored in the OCaml representation of an existential quantification. As long as we combine the first and second child nodes, we can obtain the first sub-expression of the existential quantification, a variable.

In the implementation, we use the compact representation of Chiron expressions. A node which is denoted by a path is not necessarily the root of a proper expression. We use expression components instead of sub-expressions to describe the subtrees of a node in a syntax tree. In the example above, the existential quantifier has three components, **symbol** x, **ctype** α, and **formula** B.

In the simplification process, only proper expressions can be simplified. However, a path can point to an expression component that is not a proper expression such as a **symbol**. We use another separate function to calculate the immediate sub-expressions of a given expression. Sub-expression and expression component are two different concepts which can be used by both the simplifier and other applications.

4.4 The Path Module

In the path module, we define a record type with one field, which is a integer list, to store the path information. The reason why we use a record type to wrap up a list is that we want to create a path type which is distinguished from a list. This record type also serves as a prototype of a path type which is structural and extendable. We can include other information in it in the future by adding new fields. In this thesis, other new types such as rule and context are also defined as record types. The most useful function in this module is called `expressionPath` which takes two arguments, an **sexpression** and a path. This function returns an **sexpression** which is denoted by the given path in the given expression.

A path may not denote any vertex in some cases. We say a path is *in the expression* when this path denotes some expression component. Otherwise, we say this path is *out of the expression.* If we get an expression component which is not a proper expression, we use **improper** to wrap it because **improper** is a subtype of **sexpression**. In the previous example of an existential quantification, `FExists` (x, α, B), if we give a path $\langle 0 \rangle$, `expressionPath` will return an `ISym "exists"`. If we give a path $\langle 4 \rangle$ or $\langle 1, 2 \rangle$, `expressionPath` will raise an `"out of the expression"` exception.

One special case for expression components is operator application. There are three kinds of operator applications in the implementation: `OpApp`, `FOpApp`, and `TOpApp`. They are of type **term**, **formula**, and **ctype** respectively. In Chiron, an operator application is defined as

$$(\mathsf{op\text{-}app}, (\mathsf{op}, o, k_1, \ldots, k_{n+1}), e_1, \ldots, e_n).$$

In our Chiron implementation, a term operator application is written as

`OpApp` (s, kdl, α)

where s is the **symbol** of operator, kdl is a **kinded** list, and α is a **ctype**.

The use of **kinded** lists guarantees that the operator application has the correct number of arguments and every argument matches the corresponding **kind**. However, when `expressionPath` deals with an operator application, it will work on the official notation of the expression. The first expression component of `OpApp` (s, kdl, α) is `ISym OpApp`. The second expression component of `OpApp` (s, kdl, α) is an **operator** which contains both a name and a **kind** list. The third expression component of `OpApp` (s, kdl, α) is the first argument of the operator application if the operator is not 0-ary. There are two reasons why we treat operator application as official notation. One reason is that the information of the operator including the name and `kind` list are closely related to each other. Another reason is that arguments usually need to be processed separately. Thus, instead of treating a `kinded` list as one component, it is better to retrieve the information of arguments from the `kinded` list.

Chapter 5

Contexts

Context is an important concept in simplification. It helps us build a more powerful and trustable simplifier. The three major uses of contexts are

(1) Users can add their assumptions to a context.

(2) A context can speed up the simplification process because some results can be looked up directly from the context.

(3) Contexts can be used to discharge the conditions in conditional rewrite rules.

In the following sections, we are going to introduce the main ideas about contexts and the implementation of contexts in our work.

5.1 Global and Local Contexts

In mathematical deduction or calculation, it is natural for people to make some background assumptions. For example, $x^2 > 0$ simplifies to true in a context in which $x \neq 0$. A context is a facility for managing such assumptions. What are contexts in our MathScheme project? In [11], a *context* is described as set of formulas $\Gamma =$

$\{\varphi_1, \ldots \varphi_n\}$. The formulas in a context ordinarily serve as background assumptions for mathematical activities. A formula φ is true in the context Γ if the members of Γ logically imply φ. In rigorous mathematical reasoning, contexts are necessary to guarantee the correctness of our answers. The following examples show the uses of contexts in simplification:

(1) The expression $(x-1)/(x-1)$ can be simplified to 1 only if $x \neq 1$ is known to be true. If we want a rule like $x/x = 1$, we can define it as a conditional rule which we will present in chapter 6.

(2) The formula $A \vee B \vee C \vee D \vee E$ can be immediately simplified to **true** if we know $D = \mathsf{T}$.

(3) An expression can be converted to a conditional by introducing assumptions. Thus, $|x + |3 - x||$ has the same value as the conditional $\begin{cases} 2x - 3 & \text{if } x \geq 3 \\ 3 & \text{otherwise} \end{cases}$

When people do mathematical activities, they actually have a lot of implicit background assumptions. These assumptions constitute a global context. A global context is a very large set which contains all the axioms and mathematical knowledge we know. Regarding certain expressions or certain reasoning processes, the contexts we are interested in here may contain some formulas in addition to a global context. Moreover, different places in an expression, may have different contexts. That is why the notion of a local context is introduced in [16]. A *local context* is a context at a place in an expression. For example, in expression $\mathsf{if}(A, b, c)$, the local context of b contains A while the local context of c contains $\neg A$. From the view of a syntax tree, every node of a tree has its own local context. When we traverse the syntax tree of a given expression in simplification process, we need to know the local context of the node we have encountered.

5.2 Calculation of a Local Context

As we have mentioned in the previous section, a context is a set of formulas. Consequently, the representation and management of context focuses on expression of type **formula** in Chiron. First, we use a formula to represent a background assumption in a context. Second, we usually use and recalculate context information when we deal with a formula or a formula involved expression such as a conditional term. Every formula in a context is interpreted as a true assertion. In many expressions, different places in the expression have the same local context (see the discussion below). It is easy to calculate the local contexts in these cases. In Chiron, three types of expressions have effects on the local contexts of sub-expressions. They are conditional expressions, variable binders, and logic connectives.

Conditional Expressions

Conditional expressions have a form like $\mathsf{if}(A, b, c)$ where A is a formula which serves as a condition and b and c are terms which represent the two possible values. Let the context of $\mathsf{if}(A, b, c)$ be Γ. Then the local context of A is Γ. The local context of b is $\Gamma \cup \{A\}$ and the local context of c is $\Gamma \cup \{\neg A\}$.

Variable Binders

There are eight variable binders in our Chiron implementation.

(1) Dependent function type $(\Lambda\, x : \alpha\ .\ \beta)$.

(2) Existential quantification $(\exists\, x : \alpha\ .\ B)$.

(3) Unique existential quantification $(\exists !\, x : \alpha\ .\ B)$.

(4) Universal quantification $(\forall\, x : \alpha\ .\ B)$.

(5) Function abstraction $(\lambda\, x : \alpha\ .\ b)$.

(6) Definite description $(\iota\, x : \alpha\ .\ B)$.

(7) Indefinite description $(\epsilon\, x : \alpha\ .\ B)$.

(8) Class abstraction $(\mathrm{C}\, x : \alpha\ .\ B)$.

In the body of a variable binder, the variable symbol x is bound and another x which appears outside the variable binder is invisible. Let x be a bound variable. When we calculate the local context of the body of a variable binder, we need to remove all the formulas in which x is free from the context. For example, we have an expression $(\forall x : \mathsf{nat}\ .\ x > 5)$ where nat is natural number. Furthermore, suppose we also have $x > 8$ in our context. If we do not remove $x > 8$ from our context when we deal with the body of the universal quantification, we will conclude that the formula is true. However, this formula is actually false.

Let the context of $(\star x : \alpha\ .\ e)$ be Γ where $\star$ is Λ, λ, ι, ϵ, $\exists$, $\exists!$, $\forall$, or C. The local context of α is still Γ. The local context of e is $\Gamma - \{c_1, \ldots, c_n\}$ where the c_i are the context formulas in which x is free.

Logic Operations

Among basic logic operators, And, Or, and Implies have effects on the local contexts. Suppose we scan an expression in a order from left to right. In conjunctions, when we calculate the local context of some formula, all the previous formulas (formulas on the left of the current formula) will be added into context. In implication, the first formula will be added into context when we deal with the second formula. In disjunction, all the negations of the previous formulas will be added into the context.

Let the context of the logic operations be Γ. We have

(1) For $A_1 \wedge A_2 \wedge \cdots \wedge A_n$, the local context of A_1 is Γ, the local context of A_2 is $\Gamma \cup \{A_1\}$,..., the local context of A_n is $\Gamma \cup \{A_1, \ldots, A_{n-1}\}$.

(2) For $A_1 \supset A_2$, the local context of A_1 is Γ, the local context of A_2 is $\Gamma \cup \{A_1\}$.

(3) For $A_1 \vee A_2 \vee \cdots \vee A_n$, the local context of A_1 is Γ, the local context of A_2 is $\Gamma \cup \{\neg A_1\}$,..., the local context of A_n is $\Gamma \cup \{\neg A_1, \ldots, \neg A_{n-1}\}$.

5.3 Use of Context

We can design different rules to make use of context. When we try to simplify a formula F, we can first check if this formula is in the context. If F is in the context, we can immediately simplify F to T. If the negation of F is in the context, we can simplify F to F. We have a built-in rule called `inContextRule` which checks if a formula is in a given context. This rule only checks the syntactic form of formulas. For example, if we have a formula $A \wedge A \wedge \mathsf{T}$ in the context and try to simplify the formula A in an expression by applying `inContextRule`, then the simplifier does not simplify A to T although $A \wedge A \wedge \mathsf{T}$ being in the context means A is true.

Many simplification rules can use the context. We can have different versions of simplification rules for the same operator. The following examples are two implication rules written in OCaml.

(1)
```
let implication1 (input:formula)
(c:formula context) : formula = match input with
  | 'FOpApp (K K.Implies, [KDFormula f1; KDFormula f2]) ->
      if (f1 = C.falsef) || (f2 = C.truef) then 'FConstant (K K.True)
```

```
        else if (f1 = C.truef) then f2
        else if (f2 = C.falsef) then Constructors.Raw.notf f1
        else if f1=f2 then 'FConstant (K K.True)
        else input
    | _ -> input
```

(2)
```
let implication2 (input:formula)
  (c:formula context) : formula = match input with
    | 'FOpApp (K K.Implies, [KDFormula f1; KDFormula f2]) ->
        if inContext f1 c then f2
        else input
    | _ -> input
```

The first function `implication1` is designed for general simplification of implication. It does not use context information in the simplification process although it takes a context argument. The second function `implication2` is relying on the context. It simplifies the implication to the second formula if the first formula of implication is in the context. This simplification rule is a representation of the modus ponens inference rule.

5.4 Context Module

In the context module, a context is designed to be a record type which contains a list of variant types. In practical use, a context type contains a list of formulas. Some basic operations such as insertion, deletion, and searching are provided to manage contexts. The following table lists the interface functions of context module.

For general purpose, `createContext` returns a context type which contains a empty list of variant types. We can add a **formula** or **sexpression** into context.

Table 5.1: Interfaces for `context` type

FUNCTION NAME	USE
`createContext`	create an empty context list
`inContext`	check whether a given formula is in given context
`addContext`	add a new formula into context
`deleteContext`	delete one formula from context
`deleteContextList`	delete a list of formula from context

Currently, we use a **formula** list to represent contexts. `inContext` takes two arguments, a formula and a context, and returns a boolean value. If the given formula is in the given context, `inContext` returns Tand otherwise returns F. `addContext` first checks if the given formula is in the given context. If it is not, the given formula will be inserted into the given context. Thus, there is no duplication of formulas in our contexts. `deleteContext` also first checks if the given formula is in the given context. If it is, the given formula will be deleted from the given context. `deleteContextList` allows us to delete several formulas at the same time.

We have two options when we calculate the local contexts for a syntax tree. The first option is to calculate the local context of every node in a tree. Another way is to calculate the local context when it is needed. We choose the first method in our work because we need to keep track of the local context information at every node. Therefore, in the simplifier module, every simplifier takes an argument of type context which records the context information of the input expression. In the manipulation of a syntax tree, we pass the context information from the current node to its children nodes. In this way, we do not need path information to bind sub-expressions and local contexts. Moreover, we have another function which can calculate the local context of

given sub-expression. This function give us certain local context information without calculating the local context for every node of a syntax tree. More details about this function can be found in chapter 8.

Chapter 6

Simplification Rules

A simplification rule is a small unit of mathematical knowledge we use to simplify expressions. Simplification rules are the core of the simplifier. A good and efficient simplifier relies on the design of the rules it uses. This chapter is going to discuss what a simplification rule is and how the simplifier uses these rules. Then we will talk about some implementations of simplification rules.

First, let us introduce the notion of a transformer in Chiron. In [12], a *transformer* is defined as a function that takes expressions as input and returns an expression as output. A transformer is an important notion in MathScheme because many useful operations can be formalized as transformers. They are the major gears for computation in our system. The algorithmic side of a biform theory is embodied in the theory's transformers.

6.1 Representation of Rule

In simplification, rewrite rules and conditional rewrite rules are the most popular rules. They are represented by equations and conditional equations, respectively. The process of simplification is to change an expression which matches the form of the left-hand side of the equation to another expression which matches the form of the right-hand side of the equation. In our work, a simplification rule is a transformer parameterized by the background context. Thus all the rules are implemented as functions of type `'a -> context -> 'a` in OCaml where `'a` could be **sexpression**, **operator**, **ctype**, **term**, or **formula**. The simplification framework can be equipped with different kinds of rules. Rewrite rules, substitution rules, inference rules, and transformers can be used as simplification rules.

For example, we have a rule for simplifying double negation, $\neg\neg A \equiv A$. This rule is a rewrite rule. We do not need to worry about what the structure of A is. A could be a very large expression or just a simple formula constant. Moreover, in this example, we can do more work beyond the syntax aspect of the expression. We need to make sure that A is a formula which denotes a truth value so that the equation $\neg\neg A \equiv A$ makes sense. However, the type checking of OCaml already guarantees the well-formedness of the expressions because we cannot build an expression $\neg\neg A$ in our system unless A is a formula. In some other cases, type checking is performed by the simplifier explicitly. This kind of type checking is performed by some specific rules we will talk about in the example section.

At the first stage of design, the input and output of a simplification rule are both expressions. However, we need to take context into account because some rules are based on the context information. For example, how do we use *modus ponens* as

simplification rule? The *modus ponens* rule of inference is written in following form:

$$\frac{P \to Q, P}{Q}$$

We can make a simplification rule that takes both an expression and a context. When this function acquires an expression in the form $P \to Q$, it will check if P is in the context. If P is in the context, $P \to Q$ can be simplified to Q. Another conditional rewrite rule example is for the absolute value of a real number. We can write an absolute value rule like this:

$$|x| = \begin{cases} -x & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ x & \text{if } x > 0. \end{cases}$$

When the simplifier takes an expression in the form of $|x|$, the simplifier will first check if $x < 0$, $x = 0$, or $x > 0$ is in the context. The simplifier can simplify $|x|$ to a certain form according to the sign of x.

Therefore, the general simplification rule function has two arguments, an input expression and a context. For the rules which have nothing to do with context, the context argument does not have any effect in the body of function.

6.2 Application of Rules

In a complicated simplification process, a set of rules will be applied. Our framework takes an expression and traverses the syntax tree of this expression. At every node, the framework applies the simplification rules to the expression represented by this node. In this procedure, the simplifier tries many rules to simplify an expression. How

does the simplifier select a rule? The easiest answer is that the simplifier tries all the rules available. Obviously, this method is inefficient if the rule set is very large. How does the simplifier choose rules? Where are the simplification rules stored? These questions are related to the organization of rules. In our framework, we need some facilities to manage rules.

6.2.1 Built-In Rule Table

Our simplification framework has a built-in rule table implemented as a hash table. By using a hash table, we can look up rules via an index. Rules are categorized according to the constructor used to build the expression. Every rule's index is the keyword corresponding to its proper expression category. For example, we have a rule for conditional expression,

$$\texttt{If}\ (F, a, b) = \begin{cases} a & \text{if } F = \mathsf{T}, \\ b & \text{if } F = \mathsf{F}. \end{cases}$$

The index of this rule is `If`. When we try to simplify a conditional expression, we look up all the rules related to `If` and try applying them to the expression. Although the simplifier still needs to try a list of rules, the range is narrowed down a lot. The order of the rules taken from rule table is random. If the simplification process is confluent, it does not matter what order we apply rules in. Otherwise, the order in which the rules are applied may effect the final result of simplification.

6.2.2 User Defined Rule List

Apart from using the built-in rule table, users can create their own rule list and let the simplifier use this rule list. Compared with a built-in rule table, a user defined rule list is a flexible approach. Users can put any number of rules in this list. The

order of these rules is defined by user as well. Thus, the simplifier will apply these rules in the particular order specified by the user. This approach is useful when the user has a plan to simplify an expression. It allows a user to experiment with different combinations of rules when simplifying expressions. Normally, users who define their own rule lists want to use built-in rules as well because the built-in rules automatically handle the basic and "trivial" simplification processes. In the mode of using the user-defined rule list, our framework let users to choose whether the simplifier uses built-in rules or not. If the users choose to use both user-defined rule lists and built-in rules, they can simplify the expressions with built-in rules before trying the user-defined rule list, or first apply the user-defined rule list, and then use built-in rules. The users can specify the order.

The shortcoming of a user-defined rule list is its reusability. We do not have a good mechanism to manage these rule lists. Every rule list is designed for a certain family of expressions and may not be suitable for others. These rule lists are defined by the user when the simplifier is called. They are not stored in the system.

6.3 Rule Module

In our system, a simplification rule is a record type with two fields. The first field is the *key* which is of type string. The second field is a function that takes an expression and a context as input and returns an expression as output. There are three interface functions for the `rule` type.

The rule table is defined as hash table from type string to type simplification function. The key of a rule is also used as the key of this rule in hash table. We have several rule tables because the signatures of the simplification functions differ. The

Table 6.1: Interfaces for the `rule` type

FUNCTION NAME	USE
`getKey`	returns the key of a rule
`getRule`	returns the simplification function of a rule
`createRule`	creates a new rule

major three rule tables are the type table, term table, and formula table which contain rules for simplifying expressions of type **ctype**, **term**, and **formula** respectively. The interface functions for the rule table module are listed as follows.

Table 6.2: Interfaces for the rule table module

FUNCTION NAME	USE
`createTable`	creates a empty rule table
`addRule`	adds a rule into a rule table
`deleteRule`	deletes one rule according the given key
`deleteAll`	deletes all rules with the given key
`findAll`	returns a list of rules with the given key

End users and developers can use the interface functions of rule module to manage rule tables, create new rules, and add new rules into their corresponding rule tables. A powerful and efficient simplifier is heavily dependant on the rules it has available. It is hard to put all useful rules in our framework at once. At this stage, some basic rules are defined and added into rule tables. The rule set will be gradually improved by adding new rules.

6.4 Examples of Simplification Rules

In our framework, rules for basic logic operators are implemented. We have following rules:

$(A \equiv A) \equiv \mathsf{T}$

$(\mathsf{T} \vee A) \equiv \mathsf{T}$

$(A \vee \mathsf{T}) \equiv \mathsf{T}$

$(\mathsf{F} \vee A) \equiv A$

$(A \vee \mathsf{F}) \equiv A$

$(A \vee A) \equiv A$

$(\mathsf{T} \wedge A) \equiv A$

$(A \wedge \mathsf{T}) \equiv A$

$(\mathsf{F} \wedge A) \equiv \mathsf{F}$

$(A \wedge \mathsf{F}) \equiv \mathsf{F}$

$(A \wedge A) \equiv A$

$(\neg\neg A) \equiv A$

$(\mathsf{F} \supset A) \equiv \mathsf{T}$

$(\mathsf{T} \supset A) \equiv A$

$(A \supset \mathsf{T}) \equiv \mathsf{T}$

$(A \supset \mathsf{F}) \equiv \neg A$

$(A \supset A) \equiv \mathsf{T}$

In the Chiron implementation, the formulas above which contain logic connectives are formalized as formula operator applications. For example, $A \wedge B$ is represented as `FOpApp` (And, A, B) in our system, where keyword `FOpApp` means formula operator application. Therefore, every A in the equations above must be defined and must be a formula. A rule which is related to operator $\wedge$ has a key of string type like `"FOpApp-And"` in formula rule table. One rule function can apply several rewrite

rules of the same form. For example, we can write one rule to deal with all rewrite rules related to disjunctions. Rule functions may have overlapping over some expressions. Moreover, some simplification can be done by using the context mechanism. We already talked about contexts in chapter 5.

Besides formula operator applications, we have several other rules for conditional terms, evaluations, and beta reduction.

A **conditional term** is represented as $\mathsf{if}(A, a, b)$ in Chiron. The simplification rule for conditional expression is,

$$\mathsf{if}(A, a, b) = \begin{cases} a & \text{if } A = \mathsf{T}, \\ b & \text{if } A = \mathsf{F}. \end{cases}$$

If we cannot judge the value of the condition, we just leave the expression in its original form. The key of a simplification rule for a conditional term is `If` in the term rule table.

Evaluation is a powerful facility in Chiron. It returns the semantical meaning of an expression that represents an expression. In the implementation of the Chiron type system, we have three kinds of evaluations, `Eval`, `FEval`, and `TEval`. They are used to represent expressions of the form $(\mathsf{eval}, a, \alpha)$, $(\mathsf{eval}, a, \mathsf{formula})$, and $(\mathsf{eval}, a, \mathsf{type})$ respectively. The compact notations of these three expressions are $[\![a]\!]_{\alpha}$, $[\![a]\!]_{\mathrm{fo}}$, and $[\![a]\!]_{\mathrm{ty}}$. On the other hand, quotation is used to denote the syntactic construction of an expression. In our type system, any **sexpression** can be quoted, written as (quote, e). The compact notation of quotation is $\ulcorner e \urcorner$. The simplification rule for evaluation is,

6. Simplification Rules

$[\![\ulcorner e \urcorner]\!]_k = e$

where e is eval-free and (k=type and **type**[e]), (**type**[k] and **term**[e]), or (k=formula and **formula**[e]).

When we apply this rule, we need to check if the quoted expression is eval-free and if the **kind** of quoted expression matches the **kind** of evaluation. That an expression is eval-free means there is no eval symbol in the expression that is not in a quotation. The result of evaluating a non eval free expression is undefined.

Beta reduction for terms is a method we use to simplify a function application of the form $(\mathsf{fun\text{-}app}, (\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b), a)$ in Chiron. The simplification rule for this kind of function application is to replace the variables $(\mathsf{var}, x, \alpha)$ occurring in b with a. For example, consider a function abstraction like $(\lambda x : \mathsf{nat} \,.\, x)$, which is the identity function on natural number. A function application $(\mathsf{fun\text{-}app}, (\lambda x : \mathsf{nat} \,.\, x), 11)$ can be simplified to 11. The process of simplification is actually the process of beta reduction. The simplification rule can be expressed as,

$$(\mathsf{fun\text{-}app}, (\lambda x : \alpha \,.\, b), a) = b[a \mapsto x]$$

where $b[a \mapsto x]$ means replacing the free variables x in b with the term a.
In beta reduction, substitution will be performed. Type checking and free-for checking guarantee the soundness of the operation.

Chapter 7

Simplification framework

7.1 Simplification in Chiron

Similar to simplification rules we have talked about in chapter 6, a simplifier for Chiron is a transformer parameterized by a context. Here we define what a general simplifier is in MathScheme.

Definition 2 *In MathScheme, a simplifier is a transformer that takes an expression A and returns an expression B such that A and B are semantically equivalent. The syntactic form of B is intended to be simpler than that of A.*

A transformer has both an algorithmic meaning and an axiomatic meaning. If we want to write a simplifier as a transformer in Chiron, the input and output of the transformer must be quotations. However, the simplification framework only focuses on the algorithmic aspect of the simplifier. Therefore, we do not need to worry about the representation of simplifiers in Chiron. A simplifier is implemented as an OCaml function which works on the original expressions instead of quotations. Our framework is designed to be used by other developers. Some optional arguments of the framework may be invisible to end-user while they can be useful for further

development or other applications in MathScheme. In the future, it is easy to link the representation of a simplifier and its algorithm.

A simplifier has two major tasks in our framework:

(1) Traverse the syntax tree of an expression and apply the simplification rules according to different strategies.

(2) Calculate the local context for every node and pass the information of the local context to simplifier rules.

7.2 Simplification Strategies

A strategy is the way in which the syntax tree of an expression is traversed during the simplification process. At different stages, we have different choices of what the next step is. For example, where do we start our simplification process from? Starting from the top node (the root of syntax tree) or starting from the bottom nodes (the leaves of syntax tree) are two options. After we have simplified some node, whether to go down to simplify the children nodes of current node or to go up to check the parent node of current node are two other choices. What kind of strategy we should choose is dependant on the efficiency of the simplification and the results the user wants.

It is hard to say that one strategy is better than another everywhere or that some strategy is best. What is the purpose of doing simplification? Users want to find a simpler syntactic form of an expression to replace a more complex one. Should the system transform the expression to a representation which is as simple as possible? Normally, the simplest representation of an expression is the best for manipulation and comprehension. However, we cannot always find the "simplest" syntax form of

an expression, and the standard of "simplest" may be controversial. Moreover, in some cases, the "simplest" form of an expression may be preferred by the human, but it may not be the "simplest" form for a computer system (computer algorithm). For example, people may be willing to write a subtraction expression like $x - 1$ while the computer may prefer $x + (-1)$. There are other cases in which the simplest form of an expression is not the best form for a computation or deduction. For example, consider an expression like this:

$$((\exists x : \mathbf{R} \,.\, x^2 \geq 0) \supset (A \vee (B \wedge \mathsf{F}) \vee ((C \wedge \mathsf{T}) \vee (\mathsf{T} \vee \mathsf{F})))) =$$

$$((\exists x : \mathbf{R} \,.\, x^2 \geq 0) \supset (A \vee (B \wedge \mathsf{F}) \vee (C \vee (\mathsf{T} \vee \mathsf{F}))))$$

When we try to simplify it, we need to judge if the left-hand side equals the right-hand side. If we first simplify $C \wedge \mathsf{T}$ in left-hand side to C, we will find the two sides of equation are syntactically equivalent. Therefore, we know the whole expression can be simplified to T. In another case, if we first simplify the left-hand side as much as possible, we can get the simplest form of left-hand side, T. Then, we still need to simplify the right-hand side to T so that we can conclude that the final result is T. Apparently, in the latter situation, we do some work that is unnecessary. That is why we need different strategies which provide flexibility and let the users control the simplification process. A few related discussions about strategies, term rewriting, and tree traversal can be found in [14, 21].

7.2.1 Bottom-Up Strategy

Bottom-up is a simple and straightforward strategy. When we simplify an expression E, we first try to simplify the immediate sub-expressions of E. After all the immediate

sub-expressions of E are finished, E will be changed to E'. Finally, we try to simplify E'. The simplification functions which employ the bottom-up strategy are recursive. For example, consider an expression

$$(k, E_1, E_2, \ldots, E_n)$$

where k is the keyword of the expression, and the E_i are the immediate sub-expressions.

Let the simplification function be *simp* and the evaluation strategy of *simp* be call-by-value. The bottom-up simplification process is

$$simp(k, simp(E_1), simp(E_2), \ldots, simp(E_n))$$

The function goes to the leaves of the syntax tree and simplifies those leaf nodes first. Then the function goes up to simplify the nodes on upper levels until it reach the top node of expression (i.e. the root of tree).

Compared with other strategies, the bottom-up strategy is easy to implement. It works well in many situations and simplifies expressions exhaustively. However, the bottom-up strategy has a major flaw: it can do a lot of unnecessary work. For example, let $\mathsf{if}(A, b, c)$ be a conditional term. If we know the condition A is T, we can go on to simplify b directly. Conversely, if we simplify A to F, we can go to c without considering b. In the bottom-up approach, the simplifier first simplifies A, b, and c to A', b', and c'. At the end, simplifier tries to simplify $\mathsf{if}(A', b', c')$. Therefore, if b and c are very large and complex expressions, the simplifier will waste time on some unnecessary simplification.

7.2.2 Top-Down Strategy

In contrast to the bottom-up strategy, the top-down strategy starts the simplification process from the top node of the expression. At every node, the simplifier tries to

simplify the current expression first, then it tries to simplify the sub-expressions of the current expression. The virtue of the top-down strategy is its efficiency of processing some kinds of expressions. For example, let $A \equiv A$ be an expression where A is a large and complex formula. If we use the top-down strategy, we can immediately simplify $A \equiv A$ to T. We do not need to simplify A. The situation is similar for a conditional term. We first check the condition of an if expression and try to simplify the condition. When the condition can be simplified to T or F, we go to different branches directly. The major process of a top-down simplifier is represented as follows:

First, consider an expression

$$(k, E_1, E_2, \ldots, E_n)$$

where k is the keyword of expression, and the E_i are the immediate sub-expressions.

Let the simplification function be *simp* and the result of $simp((k, E_1, E_2, \ldots, E_n))$ be $(k, E'_1, E'_2, \ldots, E'_m)$. The top-down simplification process is

$$simp(k, E_1, E_2, \ldots, E_n); simp(E'_1); simp(E'_2); \ldots; simp(E'_m)$$

The top-down strategy also has a major disadvantage. It may not simplify the expression completely in one call of a top-down simplifier. For example, using the top-down strategy, we simplify $\mathsf{if}((\mathsf{F} \vee A) \supset ((\mathsf{T} \wedge \mathsf{F} \vee A) = (\mathsf{T} \wedge \mathsf{F} \vee A)), [\![\ulcorner First \urcorner]\!]_{te}, Second)$ to $\mathsf{if}(A \supset \mathsf{T}, First, Second)$. However, the final result of simplifying the example expression is *First*. The top-down strategy just does a partial simplification. To solve this problem, we can restart the simplification after each success. The place where we restart the simplification from is decided by different strategies.

7.2.3 Combination of Different Strategies

As we mentioned in previous sections, different strategies have different advantages. Finding out a universal strategy that fits for every situation is very difficult. That is why we implement our work as a framework that supports different strategies. By using this framework, users have different options at different stages of simplification.

When we traverse the syntax tree, we have two major strategies, bottom-up and top-down. After we have found an applicable rule and simplified a node, we have several choices:

- Stop after one simplification.
- Continue at the current node.
- Continue at a lower level.
- Continue at an upper level.

The first, "non-continue" approach means the simplifier does one step of simplification, then stops. This approach is useful in some cases. For example, the user does not want the simplifier to simplify too much, or the user wants to see the simplification step by step. The non-continue approach is also the basis for other strategies. In other words, a "continue" strategy is a sequence of non-continue simplifications.

The continue at an upper level strategy is what we call backtracking. It is an important complement of the top-down strategy. For instance, we have an expression $A \wedge B$. We use the top-down strategy to simplify this expression. First, we check the top node of expression, then we find rules for and. However, we can not do anything except go deeper to check the sub-expressions because no rule is applicable

at this moment. Consequently, we go to A, then we find that we can simplify A to F. At this stage, we can go back to the top node, which is $\mathsf{F} \wedge B$. The good thing is we know the final result is F without considering what B is. Should the simplifier always go back to the top node after some node is simplified? The answer is no. It is hard to say how many levels the simplifier should backtrack. A proper number of backtracking levels depends on the structure of expression and the design of the simplification rules. If the number of backtracking levels is small, the simplifier will go through the expression more "carefully". If the number of backtracking levels is big, the simplifier will go back to the top node more "quickly" so that we may need to restart the simplification several times. One level backtracking is recommended because a node is more related to its children nodes. In other words, the changes made to a node is more likely to have an effect on its parent node. Moreover, most of the simplification rules are designed to deal with some simple expression patterns. For example, a simplification rule is designed to simplify a certain kind of operator O. This rule often just checks the operands of O to determine whether a given expression can be simplified. It will not consider the sub-expressions of the operands of O.

7.3 Architecture of the Simplification Framework

The core module of our software consists of a set of simplifiers which work together to mix different simplification strategies in one framework. The two most basic simplifiers are `simp_once` and `simp_multi`. At a higher level, we build top node simplifiers from basic simplifiers. Basic simplifiers and top node simplifiers are essential for the deep simplifiers because deep simplifiers use basic simplifiers and top node simplifiers to do actual simplification at every node they visit. Deep simplifiers can be categorized into different families according to strategies. The deep simplifiers in a family use the same strategy and take different expression types as inputs. They depend

on each other and work together to fulfill a certain kind of simplification strategy. On the other hand, deep simplifiers are divided into two groups because one group uses rules from built-in rule tables while another group also uses user-defined rule lists. The deep simplifiers that only use built-in rule tables are built from top node simplifiers while the deep simplifiers that only use user-defined rule list are built from basic simplifiers. Among different top node simplifiers, `simp_top_NC` is used more frequently because most of the deep simplifiers that depend on top node simplifiers use the non-continue strategy. The following tables show the organization of our simplification framework. In these tables, `ep` is a record type which contains an **sexpression** and a path. Type `ep` is mainly used by function `findPath` which we will introduce in the later sections.

Table 7.1: Basic simplifiers

FUNCTION NAME	INPUT/OUTPUT TYPE	STRATEGY
`simp_once`	`'a`	non-continue
`simp_multi`	`'a`	continue at current node

Table 7.2: Top node simplifiers using built-in rule table

FUNCTION NAME	INPUT/OUTPUT TYPE	STRATEGY
`simp_top_NC`	**sexpression**	non-continue
`simp_top_CO`	**sexpression**	continue at current node

Table 7.3: Deep simplifiers using built-in rule table

Function Name	Input/Output type	Strategy
simp_NC	sexpression	top-down non-continue
simp_NC_repeat	sexpression	
simp_CO	sexpression	top-down continue at current node
simp_C	sexpression	top-down continue
simp_BU	sexpression	bottom-up
simp_BU_repeat	sexpression	
simp_BT_once	ep	top-down backtracking
simp_BT	sexpression	

Table 7.4: Deep simplifiers using user-defined rule list

Function Name	Input/Output type	Strategy
simp_NC_RL	sexpression	top-down non-continue
simp_NC_RL_repeat	sexpression	
simp_CO_RL	sexpression	top-down continue at current node
simp_C_RL	sexpression	top-down continue
simp_BU_RL	sexpression	bottom-up
simp_BU_RL_repeat	sexpression	
simp_BT_once_RL	ep	top-down backtracking
simp_BT_RL	sexpression	

The above tables only select one representative from every simplifier family. There are more than one simplifier in some families. Three complete tables of simplifiers are found in appendix A.

7.4 Basic Simplifiers

As the table in the previous section shows, we have two basic simplifiers. The following is OCaml source code for these two simplifiers.

```
let rec simp_once inputExpr c rules =
  let root = inputExpr in
  let i = ref 0 in
    begin
      while !i<List.length rules && root = (List.nth rules !i) root c do
        i := !i + 1
      done;
      if !i = List.length rules then
        inputExpr
      else (List.nth rules !i) root c
    end

let rec simp_multi inputExpr c rules =
  let root = ref inputExpr in
    for i = 0 to (List.length rules) - 1 do
      root := (List.nth rules i) !root c
    done;
    if inputExpr <> !root then
      simp_multi !root c rules
    else !root;;
```

These two functions have the same signatures. They take three arguments. `inputExpr` is the expression we wish to simplify. `c` is the context that the function will use in

simplification. `rules` is a list of simplification rules. All these arguments are of polymorphic type so that basic simplifiers can be applied to different types of expressions and rules. For example, if we want to simplify a formula, we can pass this formula and a list of formula simplification rules to `simp_once` or `simp_multi`. Then the simplifier will return a simplified formula if some rule is applicable. The context `c` will be passed directly to simplification rules because only the simplification rules can make use of context.

The difference between `simp_once` and `simp_multi` is for how many times they simplify an expression. One time we talk about here means an expression is simplified by applying one rule. At the beginning, `simp_once` tries to find an applicable rule in a rule list. As long as `simp_once` finds one, it will use this rule to simplify the given expression, then return the result. On the other hand, `simp_multi` always tries to simplify the given expression as much as possible. It keeps trying other rules after successful simplifications until no applicable rule can be found. In some situations, `simp_multi` could run forever.

In our framework, all other simplifiers are built on top of these two basic simplifiers. When we use built-in rule tables, we will look up related rules and put them in a list. Then we will pass this rule list to `simp_once` or `simp_multi`. If we use a user-defined rule list, we can pass the rule list directly to `simp_once` or `simp_multi`. In our work, most of the simplifiers are based on `simp_once`. At every node of a syntax tree, the simplifier tries to simplify the sub-expression represented by this node one time, then the simplifier will return the result or go to other nodes after a success. The advantage of using one-step simplification is that we can have more control in the simplification process.

7.5 Top Node Simplifiers

The Top node simplifiers only try to simplify the top node (or root node) of an expression. It means they just check the constructor of an expression at the top level and they do not consider the sub-expressions of this expression. We have two groups of top node simplifiers. One group uses non-continue strategy which is based on `simp_once`. Another group applies continue at current node strategy which is based on `simp_multi`. Top node simplifiers use built-in rule tables. What these simplifiers do is to look up rules from rule tables and pass them to the basic simplifiers. Rules in rule tables are categorized by the constructors of expressions. Since rule tables use strings as indices, top node simplifiers convert the constructor of an expression to a string. Then simplifiers call the searching function `findAll` provided by the rule module to obtain a rule list that contains the rules having the same index. Operator application is a special case. We need to use both the constructor and the operator name as the look up key.

7.6 Deep Simplifiers

Unlike top node simplifiers, deep simplifiers go through the syntax tree of a given expression and try to simplify the sub-expressions. The three major activities of deep simplifiers are:

(1) Traversing the syntax tree of an expression according to different strategies.

(2) Applying top node simplifiers or basic simplifiers to different nodes in a syntax tree.

(3) Calculating local contexts for every node in a syntax tree.

Depending on what the rule source is, all deep simplifiers are divided into two groups. The simplifiers which employ the same strategy in two groups work in a similar way except for the lower-level functions they use. The deep simplifiers that use built-in rule tables are based on top node simplifiers. If we do not use built-in rule tables, the deep simplifiers will not call top node simplifiers. `simp_once` and `simp_multi` will be used directly to process the user-defined rule list. In this section, we will put emphasis on the deep simplifiers using built-in rule tables.

Before we talk about different kinds of deep simplifiers, we will introduce an important function called `findPath` because it is the essential part of several deep simplifiers. The major work of `findPath` is to find out the first sub-expression which can be simplified in an expression and return the path of this sub-expression. What does "first" mean here? The `findPath` traverses the syntax tree of an expression in order from top to bottom and from left to right. It is actually a preorder in depth-first traversal. In this order, `findPath` tries to apply top node simplifier to every node it visits. As long as some node can be simplified, `findPath` will simplify this node and return the simplified expression and the path of this node. This path information can be used in backtracking and non-continue strategy. Meanwhile, `findPath` calculates the local context for every node in a syntax tree as well. We define a record type called `ep` which contains two fields, `expr` and `ll`. The `expr` is of type **sexpression**. The `ll` is an integer list used to store the path. For example, consider the expression if $(A \wedge \mathsf{T}, b, c)$. First we make a variable v of `ep` type. Let v be $\{\texttt{expr} = \text{if}(A \wedge \mathsf{T}, b, c); \texttt{ll} = [\,]\}$. Passing v and an empty context to `findPath`, we obtain the result, $\{\texttt{expr} = A; \texttt{ll} = [1]\}$.[1] A is the simplified equivalent of $A \wedge \mathsf{T}$. [1] is the path of $A \wedge \mathsf{T}$. If no sub-expression can be simplified in the given expression, the output of `findPath` is the same as input. If the top node of an expression is simplified, the

[1] [1] here is not a reference. It is the representation of a list containing 1 in OCaml

output path is empty list. In previous example, let $v = \{\texttt{expr} = \text{if } (\mathsf{T}, b, c); \texttt{ll} = []\}$. Then the result of applying `findPath` to v is $\{\texttt{expr} = b; \texttt{ll} = []\}$. The empty path represents the top node of an expression.

7.6.1 Top-down Continue

The simplifier that employs the top-down continue strategy is called `simp_C`. The `simp_C` is based on a set of simplifiers which use the same strategy but take different expression types as inputs. Let us use `simp_term_C` as an example to illustrate the work process of this kind of simplifiers. The `simp_term_C` first calls top node simplifiers to simplify the input term. Then, `simp_term_C` will go to the sub-expressions of this term and try to simplify them. Therefore, no matter whether the node can be simplified or not, `simp_term_C` will continue simplifying its children nodes.

The `simp_term_C` takes two arguments, a term and a context, and returns a term. In a simplification process, when `simp_term_C` goes to the sub-expressions of a given term, the function will calculate the local contexts for the sub-expressions. Meanwhile `simp_term_C` will recursively call itself and use the local contexts to simplify the sub-expressions. In our framework, the calculations of local contexts are done by deep simplifiers.

7.6.2 Top-down Non-Continue

Top-down non-continue strategy means traversing the syntax tree of an expression from the top node and stopping simplification process after some node is simplified. `simp_NC` is based on `findPath`. After some sub-expression is simplified by `findPath`, `simp_NC` replaces the old sub-expression with the simplified one, then returns the result. The following is the OCaml source code of `simp_NC`:

```
let simp_NC (inputExpr: sexpression) c : sexpression =
  let rootep = {expr=inputExpr;ll=[]} in
    let resep = findPath rootep c in
      replace inputExpr {p=resep.ll} resep.expr
```

The `simp_NC` does only one time simplification in one call. If we want to use it to simplify an expression completely, we need to apply `simp_NC` for several times. That is why we make `simp_NC_repeat`. The `simp_NC_repeat` use `simp_NC` to simplify the given expression, then pass the result to `simp_NC` again. Users can specify how many times the simplification repeats for.

7.6.3 Top-down Continue at Current Node

The `simp_CO` starts from the top node of an expression and goes down to the children nodes if current node cannot be simplified. As long as a node can be simplified, `simp_CO` will stay at this node and try to simplify it as much as possible. The `simp_CO` depends on `findPath` as well. Similarly, after `findPath` finds an applicable rule for some sub-expression, `simp_CO` will use `simp_top_CO` to simplify this sub-expression, then return the result. The `simp_CO` performs well on some kinds of expressions such as double negations.

7.6.4 Bottom-up

The `simp_BU` uses the bottom-up strategy. There is a group of simplifiers which work together to support `simp_BU`. These simplifiers work on different types of expressions and calculate local contexts for the sub-expressions. The bottom-up simplifiers traverse syntax trees in postorder and use non-continue top node simplifiers to do simplifications at every node.

Normally, `simp_BU` simplifies a given expression exhaustively. However, in some cases, it does not because we use non-continue top-node simplifiers at a low level. Therefore, `simp_BU_repeat` is provided to repeat the bottom-up simplification. The `simp_BU_repeat` works in a similar way as `simp_NC_repeat` does.

7.6.5 Top-down Backtracking

Backtracking is a good match for top-down strategy simplifiers. In top-down traversal, if some node of a syntax tree is simplified, the simplifier will go back several levels to check its ancestor node. Some nodes may be visited by the simplifier several times during the simplification process. Users can specify how many levels the simplifier backtracks for. If the simplifier always go back to the top node of an expression, it is the same as applying `simp_NC` repeatedly. In a word, backtracking helps top-down simplifiers simplify the given expression more.

We wrote two functions to fulfill the backtracking mechanism. One is `simp_BT_once` and another is `simp_BT`. `simp_BT_once` takes four arguments, an **sexpression** *inputExpr*, a context c, a path p, and an integer i. It returns an `ep` type. The `simp_BT_once` first goes to the sub-expression e that is denoted by the given path p and tries to simplify e. If e is simplified to e', `simp_BT_once` will return the whole expression with e replaced by e' and the path, that is i levels up from p. If e cannot be simplified, `simp_BT_once` will go down and continue trying to simplify the children nodes of e until some children node e_1 is simplified. Let e_1 be simplified to e_1'. `simp_BT_once` will return the whole expression with e_1 replaced by e_1' and the path, that is i levels up from the path of e_1.

The `simp_BT` is based on `findPath` and `simp_BT_once`. It is a recursive function.

The simplification process moves between different nodes in a syntax tree and stops only if the process goes back to the top node and no sub-expression is changed. In our framework, `simp_BT` is guaranteed to terminate. The `simp_BT` does not always simplify expressions exhaustively. Therefore, we need to repeat it sometimes.

7.7 The General Simplifier

7.7.1 The Signature of General Simplifier

The most general simplification function is named `general_simplifier`. Its signature is:

```
?c:context ->
?m:mode ->
?i:int ->
?path:path ->
?rl:rule list ->
?flag:flag ->
inputExpr:sexpression -> sexpression
```

`inputExpr` is the only required parameter in this function, while other parameters are optional. The type of input and output expressions is **sexpression**. By passing different values to the optional parameters, the users can customize their own simplifier. Hence, `general_simplifier` is actually a customizable simplifier. The meanings of parameters of `general_simplifier` are listed as follows:

- `?c`, the context, is an optional argument. Its default value is an empty context.
- `?m`, the mode, is an optional argument. It specifies the strategy being used

in simplification. Its default value is **TDNC**. **TDNC** is the top-down non-continue mode we will talk about later.

- `?i`, the count, is an optional argument. It gives the number of backtracking levels ($-1 \leqslant$ `i`). In addition, `?i` can be used to specify how many times for which the user wants to apply the simplifier repeatedly in some modes ($0 \leqslant$ `i`). Its default value is 0.

- `?path`, the path, is an optional argument. It specifies where the simplification process starts from. Its default value is the empty path which means the simplification starts from the top node of an expression.

- `?rl`, the rule list, is an optional argument. It is bound to user-defined rule list. Its default value is the empty list which means the simplifier will use only the built-in rule tables.

- `?flag` is an optional argument used to specify the use of built-in rules when the simplifier takes user-defined rule list. Its default value is `Never` which means the built-in rules will not be used at all. Two other values of `flag` are `Before` and `After`. The value `Before` means trying built-in rules before applying user-defined rule list while `After` means using built-in rules after applying user-defined rule list.

- `inputExpr`, the input expression, which must be given, is the **sexpression** that the user want to simplify.

7.7.2 Modes of General Simplifier

One of several strengths of `general_simplifier` is that different modes can be chosen as simplification strategies. We defined 6 modes for our system. They are **TN**, **TDC**,

TDNC, BU, TDNCR, and BUR.

- **TN** means the function only simplifies the top node of an expression. This mode implements the non-continue approach. In other words, as long as the top node of expression can be simplified by using some rule, the function will simplify this expression by applying that rule once, then the simplification process will stop. If no applicable rule can be found for the top node, the simplification process will stop as well. Among several optional arguments, `?c`, `?rl`, and `?flag` are permitted in this mode.

- **TDC** is the top-down continue strategy. It means the simplification process goes from the top node to the bottom nodes. When the function checks a node (we call it n), whether n can be simplified or not, the simplification process will continue. The next step of the function depends on the optional argument `?i`. If `?i` is -1, it means the function will go to check the children nodes of n. If `?i` is 0, it means the function will try to simplify n again and again until n can not be simplified anymore. If `?i` is greater than 0, it means the function will go back to check the ancestor nodes of n. If `?i` is 1, the function will backtrack 1 level and check the parent node of n. If `?i` is 2, the function will backtrack 2 levels and check the grandparent node of n. If `?i` is infinity (in our work, the maximum integer), it means the function will always go back to the top node of the expression and start simplification from top node again. Generally, if the number of backtracking levels is greater than the depth of syntax tree representing the expression, the function will go to the top node and start again. In our framework, `?i` only takes one negative number -1. It means that the simplification goes down level by level. If the simplification goes down several levels every time, it will skip many nodes and stop at the bottom nodes. Working in this way, the simplifier is random and useless. In this mode,

`?c`, `?i`, `?path`, `?rl`, and `?flag` are legal.

- **TDNC** is the top-down non-continue approach. The function traverses the syntax tree of expressions from the top node down to the bottom nodes. As long as some node is simplified, the simplification process will stop. At most, only one node will be simplified once in this mode. Optional arguments, `?c`, `?path`, `?rl`, and `?flag` are permitted.

- **BU** means the bottom-up strategy. The function first simplifies the bottom nodes, then go up to simplify their parent node, grandparent node and so on. At the end, the top node of expression will be simplified. Every node will be simplified at most once. In this mode, `?c`, `?path`, `?rl`, and `?flag` are permitted.

- **TDNCR** is the top-down non-continue repeat strategy. It means the framework applies top-down non-continue simplifier repeatedly to the expression intended to be simplified. Optional argument `?i` means the simplifier will be applied $i+1$ times. If `?i` is 0, the framework applies simplifier once, which is equivalent to **TDNC**. In this mode, optional arguments, `?c`, `?i`, `?path`, `?rl`, and `?flag` are legal.

- **BUR** means bottom-up repeat strategy. It is similar to **TDNCR** except the framework applies bottom-up simplifier repeatedly in this mode.

7.7.3 Work of General Simplifier

We have already introduced the signature of `general_simplifier` and the meanings of all arguments in the previous sections. Different combinations of optional arguments give `general_simplifier` different tasks. In this section, we are going to introduce what underlying simplifiers are called when different arguments are given.

The work of `general_simplifier` is mainly based on mode argument `?m`. The default value of `?m` is **TDNC**. When we choose to use a user-defined rule list, we need to specify the argument `?flag` to decide whether the simplifiers use the built-in rule tables as well. Different modes and corresponding simplifiers are listed as follows. We assume that `?flag` is `Never`.

TN means top node only simplifier. If rule list argument `?rl` is not specified, `simp_top_NC` is called. Otherwise, `simp_once` is called.

TDC mode cooperate with optional argument `?i`. First, we assume `?rl` is empty. If `?i` is -1, `simp_C` is applied. `simp_C0` is applied when `?i` is 0. If `?i` is greater than 0, `simp_BT` is applied. In another situation, when `?rl` is given, `simp_C_RL`, `simp_C0_RL`, and `simp_BT_RL` are called.

TDNC is top-down non-continue strategy. If rule list argument `?rl` is not given, `simp_NC` is used to do simplification. Otherwise, `simp_NC_RL` is used.

BU means bottom-up approach. In this mode, `simp_BU` is called when `?rl` is not given. Otherwise, `simp_BU_RL` is called with `?rl` given.

TDNCR represents top-down non-continue repeat strategy. `?i` is used to specify the times of repeating. If rule list argument is empty, `simp_NC_repeat` is called. The repeat function with rule list argument is `simp_NC_RL_repeat`.

BUR is bottom-up repeat simplifier. It works in a similar way as **TDNCR** simplifier does. The `simp_BU_repeat` is applied when `?rl` is not specified. Otherwise, `simp_BU_RL_repeat` is applied.

If argument `?flag` is not `Never` in above modes, both the simplifiers that employ the same strategy but use different rule sources are applied to the given expressions. Let us use mode **TN** as an example, If `?flag` is `Before` or `After` when `?rl` is given, `simp_top_NC` is applied before or after `simp_once`, respectively.

If optional argument `path` is not specified, its default value is empty list. In this case, the simplification process will start from the top node of a given expression. If `path` is given, only the sub-expression which is denoted by the given path will be simplified. The original sub-expression will be replaced with the simplified one.

7.8 Tests

We did some tests for our simplification framework. The goal of testing is to verify the results of simplifications and to measure the efficiencies of different strategies. However, at current stage, there are only a few operators and simplification rules are defined. We do not have enough samples to do many tests. Therefore, the results of our tests are not conclusive because of a lack of coverage. In our testing, the following kinds of expressions were used as samples:

- Logic connectives such as $\wedge$, $\vee$, $\supset$, $=$, $\neg$.
- Conditional terms.
- Evaluations.
- Function applications.

The results of simplification are correct in our tests. When we apply different simplifiers to the same expression, the results also reflect the uses of different strategies.

We tested some simple cases of using contexts in simplification and the results are correct as we expected as well. All the tests we did illustrate that our framework and the simplification rules we designed work correctly. Furthermore, we used a timing function to measure the running time of different simplifiers when they deal with the same expression. First, we built some large but simply structured expressions. A simply-structured expression here means an expression with repeated structure. In several tests, the running time of the bottom-up strategy is less than that of the top-down strategy even though the structures of some expressions are theoretically suitable for top-down strategy. However, one factor we noticed is that our rules are simple. It means the most of running time is not spent on the application of simplification rules but traversal of the syntax tree of an expression. Then we made some time consuming rules and used the same sample and testing method again. We found that the top-down strategy beat bottom-up strategy in some cases.

Chapter 8

Some Tools

In this chapter, we will introduce some useful functions for our system. It is easier to manage and reuse these tools if we put them in a module. This module is called `Sometools`. It contains not only support tools for simplification but also some implementations of Chiron operators.

8.1 immediateSubExpressions

In path module, we use expression components to describe the structure of a syntax tree. We cannot obtain a sub-expression via path information sometimes. `immediateSubExpressions` provides a way of deconstructing an expression that is different from using expression components. This function retrieves the immediate sub-expressions from a given **sexpression**. It returns a list of **sexpression** which contains all the immediate sub-expressions in a left-to-right order. For example, `Constant` S does not have any immediate sub-expressions. The conditional term `If` (A, b, c) has three immediate sub-expressions, formula A, term b, and term c. Existential quantification `FExists` (x, α, B) has two immediate sub-expressions instead of three. In the expression, x is a symbol and α is a type. We put them

together to make a variable `Var` (x, α), which is the first immediate sub-expression. The second immediate sub-expression is a formula B.

Improper expressions do not have sub-expressions. If we apply `immediateSubExpressions` to a value of type **improper**, an exception will be raised.

8.2 is-eval-free

is-eval-free is a defined operator in Chiron which checks if an expression is free of evaluations[1]. When we try to simplify an expression like $(\mathsf{eval}, (\mathsf{quote}, e), k)$, we need to first check whether e is eval-free, then check whether e and k are good evaluation arguments. Let us use $(\mathsf{eval}, (\mathsf{quote}, e), \mathsf{type})$ as an example. Only if e is eval-free and a good evaluation argument, can $(\mathsf{eval}, (\mathsf{quote}, e), \mathsf{type})$ be simplified to e. We will talk about good evaluation arguments in next section.

The implementation of is-eval-free is according to the defining axioms of eval-free checker in [8]. The function `is_eval_free` takes an **sexpression** as input and returns a boolean value. It recursively checks the sub-expressions of the given expression. Only if all the sub-expressions are eval-free, is the whole expression eval-free. For example, we have an expression

$$(k, E_1, E_2, \ldots, E_n)$$

where k is the keyword of the expression, and E_i is its immediate sub-expression.

Let the eval-free checker be is-eval-free. The result of is-eval-free$((k, E_1, E_2, \ldots, E_n))$ is

[1] An expression e is *eval-free* if all occurrences of the symbol eval in e are within a quotation

- F if $k =$ eval
- T if $k =$ quote
- is-eval-free(E_1) $\wedge$ is-eval-free(E_2) $\wedge \ldots \wedge$ is-eval-free(E_n) otherwise

We can not check if an improper expression is eval-free. Thus, applying `is_eval_free` to an improper expression raises an exception.

8.3 Good Evaluation Arguments

In the original definition of evaluation, we need to use a value of type **kind** to differentiate the types of evaluations. The expression to be evaluated must match the **kind** of the evaluation operator. Therefore, a good evaluation arguments checker is important in the simplification of evaluation.

However, in our implementation, We have three kinds of evaluations, `Eval`, `FEval`, and `TEval`. They represent (eval, e, α), (eval, e, formula), and (eval, e, type) respectively. Therefore, we need to check whether e is a **term** in the simplification of `Eval` (e, α), whether e is a **formula** in the simplification of `FEval` (e), and whether e is a **ctype** in the simplification of `TEval` (e).

The function gea is a partial implementation of the Chiron operator gea. It takes two arguments, an **sexpression** and a **kind**. Function gea returns T if the given **sexpression** matches the given **kind**. Now, we have `is_eval_free` and gea so that we can write the simplification rule for evaluations.

8.4 replace

`replace` is an important function that is used in simplification. It takes three arguments, **sexpression**, **path**, and **sexpression**. Let us pass e_1, p, and e_2 as arguments to `replace`. `replace` will replace the sub-expression which is denoted by p in e_1 with e_2. For example, suppose we have

$e_1 =$ `If` (A, b, c)

$p = \langle 2 \rangle$

$e_2 = d$

where d is a **term**.

Then, `replace` e_1 p $e_2 =$ `If` (A, d, c).

If the path we give is empty, the whole expression will be replaced. In the example above, if $p = \langle\rangle$, e_1 and e_2 remain unchanged, then `replace` e_1 p $e_2 = d$. In the replacing process, type checking is performed. We will use the example above again. Let $p = \langle 1 \rangle$, e_1 and e_2 remain unchanged, then `replace` e_1 p e_2 will raise an exception because d is a **term** and A which is denoted by $\langle 1 \rangle$ is a **formula**. Replacing a **formula** with a **term** does not make any sense.

8.5 localContext

`localContext` is a function used to calculate the local context. This function takes three arguments, an **sexpression**, a **context**, and a **path**. It returns the local context of the sub-expression which is denoted by the given path in the given **sexpression**. Giving an empty path means calculating the local context for the root node. In this case, the output of `localContext` is the same as the input context.

In our framework, we calculate local context for every node of a syntax tree

during the simplification. Every simplifier takes a context argument used to store the context information of the input expression. Therefore, most simplifiers do not call `localContext` to calculate local contexts for sub-expressions. However, `localContext` is a useful tool of context module. It is convenient for users who want to obtain local context information in other operations.

Chapter 9

Conclusion and Future Work

We have developed a simplification infrastructure for Chiron expressions. It is a platform that developers can use it to experiment with different strategies and approaches for doing simplification. It also provides several basic components that developers can use in future developments. Based on this framework, developers can improve the simplifier by adding new simplification rules. They also can combine different strategies and design the simplification process to make their own customized simplifier.

The two major elements in the design of our simplifier are rules and strategies. We have developed some simplification rules and strategies for our framework. The rule module is a tool we use to manage simplification rules. We can add new rules, modify, or delete existing rules via the interfaces of rule module. Strategies refer to how to traverse a syntax tree and perform actions after every successful simplification. Strategies are divided into two groups, top-down and bottom-up. They have their own advantages when they deal with different expressions. Top-down is the default strategy we use in our simplification. A backtracking mechanism improves the capability of the top-down strategy. In practical use, end users or developers

choose various strategies according to the expressions they want to simplify and the simplification rules they use.

The path module is useful in simplification as well as in other applications. We can easily navigate sub-expressions and manipulate sub-expressions by using paths. Replacement of a sub-expression and the calculation of a local context rely on path information. The same mechanism of using paths can be applied to the application of transformers as well.

The notion of a context is an important facility that we use to make our simplifier more powerful. In our simplification infrastructure, local contexts are calculated when the syntax trees of expressions are traversed. The context module provides interface functions that we can use to manage context information. With contexts, we can put assumptions in our deduction and computation rules. Furthermore, contexts help speed up simplification in some cases.

Simplification is not a new topic in MMSs. Many simplifiers have already been implemented in other mathematical softwares. Our work is inspired by other contemporary systems as well. Ideas like context are employed in some mathematical software such as IMPS [10] and Isabelle [20]. The feature of our framework is its flexibility. Several strategies are provided so that users can choose a proper one from them. Simplification rules is another open module that users can develop in the future.

In the future, we need to design more rules and add them into our simplification framework. Custom strategies will be designed to increase the power and efficiency of simplifier. In the current work, rules are selected randomly from the built-in rule table. If the simplification process is not confluent, the choice of rules or the order of

applying rules may affect the final result. Therefore, a more sophisticated selector is needed. The implementation of context module can be improved. The simplifier now just checks whether a given formula is in the context when people try to use contexts in simplifications. A new approach of using context should be more reasonable. We can first simplify all formulas in context before we search. Our simplification infrastructure is designed to be used by future developers because the rules are written in OCaml. With a proper interface, the end users can create their own rules in the future. For example, we can make some functions that automatically transform an equation into a rewrite rule or transform a theory into a more complex rule. Moreover, we hope that our simplifier can have a tracker that stores a simplification process in some form. The tracker can display a simplification step by step when users want to see it.

Appendix A

Tables of Simplifiers

Table A.1: Basic simplifiers

Function Name	Input/Output type	Strategy
`simp_once`	`'a`	non-continue
`simp_multi`	`'a`	continue at current node

Table A.2: Top node simplifiers

Function Name	Input/Output type	Strategy
simp_ top_ type_ NC	**type**	non-continue
simp_ top_ term_ NC	**term**	
simp_ top_ formula_ NC	**formula**	
simp_ top_ kind_ NC	**kind**	
simp_ top_ kinded_ NC	**kinded**	
simp_ top_ NC	**sexpression**	
simp_ top_ type_ C0	**type**	continue at current node
simp_ top_ term_ C0	**term**	
simp_ top_ formula_ C0	**formula**	
simp_ top_ kind_ C0	**kind**	
simp_ top_ kinded_ C0	**kinded**	

Table A.3: Deep simplifiers using built-in rule table

Function Name	Input/Output type	Strategy
simp_ NC	**sexpression**	
simp_ NC_ repeat	**sexpression**	
simp_ C0	**sexpression**	
simp_ type_ C	**type**	top-down continue
simp_ term_ C	**term**	
simp_ formula_ C	**formula**	
simp_ kind_ C	**kind**	
simp_ kinded_ C	**kinded**	
simp_ operator_ C	**operator**	
simp_ C	**sexpression**	
simp_ type_ NC	**type**	bottom-up
simp_ term_ BU	**term**	
simp_ formula_ BU	**formula**	
simp_ kind_ BU	**kind**	
simp_ kinded_ BU	**kinded**	
simp_ operator_ BU	**operator**	
simp_ BU	**sexpression**	
simp_ BU_ repeat	**sexpression**	
simp_ BT_ once	**something**	top-down backtracking
simp_ BT	**sexpression**	

Table A.4: Deep simplifiers using user-defined rule list

Function Name	Input/Output type	Strategy
simp_ NC_ RL	**sexpression**	
simp_ NC_ RL_ repeat	**sexpression**	
simp_ C0_ RL	**sexpression**	
simp_ type_ C_ RL	**type**	top-down continue
simp_ term_ C_ RL	**term**	
simp_ formula_ C_ RL	**formula**	
simp_ kind_ C_ RL	**kind**	
simp_ kinded_ C_ RL	**kinded**	
simp_ operator_ C_ RL	**operator**	
simp_ C_ RL	**sexpression**	
simp_ type_ NC_ RL	**type**	bottom-up
simp_ term_ BU_ RL	**term**	
simp_ formula_ BU_ RL	**formula**	
simp_ kind_ BU_ RL	**kind**	
simp_ kinded_ BU_ RL	**kinded**	
simp_ operator_ BU_ RL	**operator**	
simp_ BU_ RL	**sexpression**	
simp_ BU_ RL_ repeat	**sexpression**	
simp_ BT_ once_ RL	**something**	top-down backtracking
simp_ BT_ RL	**sexpression**	

Bibliography

[1] Objective Caml. Home page at `http://caml.inria.fr/` (accessed August 26, 2010).

[2] Jacques Carette. Understanding expression simplification. In *ISSAC*, Santander, Spain, 2004.

[3] Bob F. Caviness. On canonical forms and simplification. *Journal of the ACM*, 17(2):385–396, 1970.

[4] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, 1991.

[5] William M. Farmer. A proposal for the development of an interactive mathematics laboratory for mathematics education. In *CADE-17 Workshop on Deduction Systems for Mathematics Education*, page pages, 2000.

[6] William M. Farmer. Biform theories in Chiron. In *Calculemus '07 / MKM '07: Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants*, pages 66–79, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] William M. Farmer. Chiron: A multi-paradigm logic. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej*

Trybulec, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 1–19. University of Białystok, 2007.

[8] William M. Farmer. Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report 38, McMaster University, 2007. Revised 2010.

[9] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An Interactive Mathematical Proof System. volume 11, pages 213–248, 1993.

[10] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 1993. Available at `http://imps.mcmaster.ca/` (accessed August 26, 2010).

[11] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. Contexts in mathematical reasoning and computation. *Journal of Symbolic Computation*, 19(1-3):201–216, 1995.

[12] William M. Farmer and Martin v. Mohrenschildt. Transformers for symbolic computation and formal deduction. In *Proceedings of the Workshop on the Role of Automated Deduction in Mathematics, CADE-17*, pages 36–45, 2000.

[13] William M. Farmer and Martin v. Mohrenschildt. An overview of a formal framework for managing mathematics. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):165–191, 2003.

[14] Paul Klint and Jurgen J. Vinju. Term rewriting with traversal functions. Technical report, ACM Transactions on Software Engineering and Methodology, 2001.

[15] Mathscheme: An integrated framework for computer algebra and computer theorem proving. Web site at `http://www.cas.mcmaster.ca/research/mathscheme/` (accessed August 26, 2010).

BIBLIOGRAPHY

[16] Leonard G. Monk. Inference rules using local contexts. *Journal of Automated Reasoning*, 4(4):445–462, 1988.

[17] Joel Moses. Algebraic simplification a guide for the perplexed. In *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 282–304, New York, NY, USA, 1971. ACM.

[18] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[19] Hong Ni. Chiron: Mechanizing mathematics in OCaml. Master's thesis, McMaster University, 2009.

[20] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[21] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[22] Stephen Wolfram. *Mathematica: A system for doing mathematics by computer (2nd ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.

[23] Jian Xu. *Mei — A Module System For Mechanized Mathematics System*. PhD thesis, McMaster University, 2008.