

CHIRON : MECHANIZING MATHEMATICS IN OCAML

By
HONG NI, B.SCIENCE

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

© Copyright by Hong Ni, August 25, 2009

MASTER OF COMPUTER SCIENCE (2009)
(Department of Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Chiron : Mechanizing Mathematics in OCaml

AUTHOR: Hong Ni, B.Science(University of Toronto, ON, Canada)

SUPERVISOR: Dr. Jacques Carette

NUMBER OF PAGES: x, 108

Abstract

Computer algebra systems such as Maple [2] and Mathematica [12] are good at symbolic computation, while theorem proving systems such as Coq [11] and PVS [9] are well-developed for creating formal proofs. However, people are searching for a mechanized mathematics system which can provide highly integrated symbolic computation and formal deduction capabilities at the same time.

My work is to design and implement the basis for a mechanized mathematics system based on a formal framework, which was previously developed as part of the MathScheme project at McMaster University. The core idea of the framework consists of the notion of a biform theory, which is simultaneously an axiomatic theory and an algorithmic theory, providing a formal context for both deduction and computation.

A mechanized mathematics system which utilizes biform theories to represent mathematics requires a logic in which biform theories can be expressed. Chiron, as a derivative of von-Neumann-Bernays-Gödel set theory, is the logic we choose for our MMS development. It is intended to be a practical, general-purpose logic for mechanizing mathematics and has a high level of both theoretical and practical expressivity compared to other logics such as Zermelo-Fraenkel (ZF) set theory and first-order logic (FOL).

The thesis presents the first stage of the development of the MMS. In particular,

the type system of the MMS has been fully established along with all necessary expression constructors for building typed Chiron expressions. Half of the work for formalizing biform theories in Chiron has been implemented by introducing the notion of name spaces, which is used for exporting the low level implementation of Chiron transformers. We have experimented with the Chiron representation for expressing the meaning formulas of Chiron transformers, in particular for boolean algebra and logical operators in the other half of the work.

Acknowledgements

I would like to first and foremost express my sincere thanks and appreciation to my supervisor, **Dr. Jacques Carette**, for his invaluable guidance and insightful comments. Without his expertise and intellectual support, the development of the MMS would not have been possible. I truly appreciated all the time and advice he gave me throughout my graduate studies at McMaster University.

I would also like to thank **Dr. William M. Farmer** who helped me understand Chiron and the notion of biform theories throughout the process of developing the MMS.

My deepest gratitude and thanks go to my family, my beloved father Chunxin Ni and mother Jianpin Pang, for their endless love, constant support and continuous encouragements.

Finally, a very special thanks goes out to my lovely and wonderful wife Jie Gao for her great understanding and patient love. I am thankful for her unconditional selfless support that allowed me to devote all my time to my professional work over years.

Contents

Abstract	i
Acknowledgements	iii
1 Introduction	1
1.1 Contents	1
1.2 Naming Convention	2
1.3 Fonts	2
1.4 Contribution	3
2 Background	4
2.1 MathScheme	4
2.2 Chiron	5
2.3 Biform Theories	6
2.4 Programming Language Choice	6
3 Goals	8
3.1 Design Goals	8
3.2 Implementation Goals	9

3.3	Design Overview	10
3.4	Implementation Overview	11
4	Overview of Implementation	13
5	Type System	18
5.1	Values & Expressions	18
5.2	Expressions in Chiron	20
5.2.1	Data Type Choice for Chiron Expression	20
5.2.2	Symbol	23
5.2.3	S-Expression	24
5.2.4	Proper S-Expression	25
5.2.5	Improper S-Expression	34
5.2.6	quoted Term	36
6	Constructors	41
6.1	Chiron Types	41
6.2	builtin Module	44
6.3	Constructors for Meaning Formulas	46
7	Name Spaces	48
7.1	Transformers in Chiron	48
7.2	Implementation Modules	50
7.2.1	Module Structure	51
7.2.2	π in <i>Chiron</i>	53
7.2.3	Examples	54

7.3	The Kernel Theory	56
7.3.1	Logicals	57
7.3.2	Basics	57
7.4	Libraries	57
7.4.1	λ -Calculus	57
7.4.2	Natural Numbers	57
7.5	Name Space Environment	58
7.5.1	Name Space Environment Introduction	58
7.5.2	Name Space Environment Components	59
7.5.3	Name Space Environment Operations	60
8	Run	64
8.1	Operator Applications	64
8.2	Validation	66
8.3	Execution	67
9	Simplify	69
9.1	Implementation Module	69
9.2	Boolean Algebra Simplifications	71
10	Beta Reduction	72
10.1	Implementation	72
10.1.1	Expression Syntax	73
10.1.2	Redex	74
10.1.3	Infinite Looping	74
10.1.4	Reduce the Redex	75

CONTENTS	vii
10.2 Tests	76
11 Biform Theory	77
11.1 Definition	77
11.1.1 Biform Theories in Chiron	77
11.1.2 Rules	78
11.2 Chiron Representation	79
11.3 Biform Theory of Peano Arithmetic	80
12 Testing : Church Numerals in Chiron	85
12.1 Chiron Representation	86
12.2 Tests for Beta Reduction	89
13 User-defined Transformers	93
13.1 Compiler	93
14 Conclusion	96
A Compact Notation	99
B Chiron Types	101
C <i>Logicals</i> Library	103
D <i>Basics</i> Library	105
Bibliography	108

List of Figures

4.1	Core Modules	14
4.2	Special Modules	16
4.3	Testing Modules	17
5.1	Translator	39
7.1	Module Structure	51
7.2	Name Space Environment	59
B.1	Chiron Types	102

List of Tables

5.1	Values & Expressions	19
5.2	type symbol	23
5.3	The keyword of Chiron.	24
5.4	type sexpression	25
5.5	Proper Expressions in Chiron	28
5.6	type proper	28
5.7	type kind	30
5.8	type kinded	31
5.9	type typ	33
5.10	type operator	33
5.11	type formula	34
5.12	type term	34
5.13	type unknown	34
5.14	type quoted	36
6.1	type t	42
6.2	Constructors for type t	42
6.3	Special Values	45

6.5	Additional Built-In Operators in Chiron	45
6.4	Built-In Operators in Chiron	46
10.1	Patterns of $E_1 \rightarrow_\beta E_2$ may involve a beta reduction	73
10.2	Substitution	75
10.3	Free Variable	75
11.1	A Series of Axioms for Defining The Properties of Natural Numbers	82
11.2	More Axioms for The Natural Numbers	83
12.1	Natural Numbers	86
12.2	Compact Notation for Church Numerals in <i>Chiron</i>	89
12.3	Testing : Church Representation in Chiron	92
A.1	Compact Notation	99
A.2	Additional Compact Notation	100
C.1	Logicals Library	103
D.1	Basics Library	105

Chapter 1

Introduction

As part of the MathScheme project, we need to design and implement a *mechanized mathematics system* (MMS) based a formal framework that integrates and generalizes symbolic computation and formal deduction. The formal framework was previously developed as the first goal of the MathScheme project. I am working on the code implementation of the MMS, based on the logic called Chiron, for my graduate work supervised by Dr. Jacques Carette.¹

1.1 Contents

We start with the Background chapter which is intended to give you a quick introduction to both the MathScheme project and the logic, Chiron, used for our mechanized mathematics system. Then, a general introduction to the notion of a *biform theory*, which is the core idea of FFMM², will be given. Chapter 3 will present both the de-

¹Address: Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario L8S 4K1, Canada. E-mail: carette@mcmaster.ca.

²A Formal Framework for Managing Mathematics [6].

sign goals and the implementation goals we want to achieve for the first stage of the development of *Chiron*. This is followed by Chapter 4, which presents an overview of the implementation for the current version of *Chiron* system, laying out a general idea about the overall design of the system in terms of modularizations.

Next, chapters 6 - 11 show the details of the implementation of the system. Then, in Chapter 12, an interesting experiment on “Church Numerals in *Chiron*” will be presented. Lastly, a small *compiler* is documented in Chapter 13. Finally, the thesis will be concluded by the Conclusion chapter.

1.2 Naming Convention

Since Chiron is the core logic which is going to be used for our mechanized mathematics system, it becomes the soul of the MMS; the term *Chiron* in italic face will be used for the rest of my paper to refer to the MMS we are implementing.

1.3 Fonts

Special fonts used throughout the thesis are listed as follows :

- **Sans Serif & Bold** - Chiron Types [such as type **symbol** and type **formula**]
- **Slanted & Bold** - Module Names [such as *types* and *constructors*]
- *Italic* - Concept Term Emphasized [such as *meaning formula* and *biform theory*]
- **Bold** - Chapter Names [such as **Goals** and **Type System**]

- Typewriter - Code Related [such as a function name “`add`” and a name of a variable appears in the code “`argument`”]

1.4 Contribution

The formal specification of the logic, Chiron, was previously documented in Dr. William M. Farmer’s paper “Chiron : A set theory with types, undefinedness, quotation, and evaluation” [5], which is our main source of documentation for our implementation of the MMS. Therefore, any definition related to the logic Chiron is quoted directly from [5], unless otherwise stated. I use phrases such as “Chiron defines ...”, “...from the Chiron paper”, in my thesis to refer to the previous work done by Dr. Farmer in his Chiron paper [5].

Our contribution is the design and implementation of a well-tested program which realizes this specification. Furthermore, this thesis documents the non-trivial design choices necessary to implement Chiron safely and efficiently.

Chapter 2

Background

2.1 MathScheme

Computer algebra systems and automated theorem proving systems are two major types of mechanized mathematics systems (MMSs). Computer algebra systems are good at symbolic computation, while theorem proving systems are well-developed for creating formal proofs. However, none of those mechanized mathematics systems can provide both highly integrated symbolic computation and formal deduction capabilities. MathScheme is a project to develop a new approach to mechanized mathematics in which computer algebra and computer theorem proving are merged without sacrificing power or soundness. The short-term goals of the MathScheme project are

- (1) Develop a formal framework that integrates symbolic computation and formal deduction.
- (2) Design and implement a MMS based on the formal framework.

The long-range goal is to build, on top of the MMS, an interactive mathematics laboratory (IML) that provides an integrated set of tools for facilitating and managing mathematical reasoning. The IML is intended to have the capabilities of both contemporary computer algebra systems and computer proving systems, and the means to formalize a wide range of mathematical knowledge.

More information about the MathScheme project can be found from its homepage, at <http://imps.mcmaster.ca/mathscheme/>.

2.2 Chiron

Chiron [5] is a derivative of von-Neumann-Bernays-Gödel set theory that is intended to be a practical, general-purpose logic for mechanizing mathematics. A theoretically expressive and practically expressive logic is required for a practical, general-purpose MMS. Traditional theories, such as ZF and NBG set theories, are designed to be used in theory, not in practice.

By integrating several reasoning paradigms, Chiron has a much higher level of practical expressivity than traditional logics. As a multi-paradigm logic, Chiron supports in an integrated manner five reasoning paradigms, namely classical, permitted undefinedness, set theory, type theory and formalized syntax [4].

A separate formal, complete presentation of the syntax and semantics of Chiron is given in [5].

2.3 Biform Theories

The notion of a biform theory was first introduced as part of FFMM, a Formal Framework for Managing Mathematics [6] developed as part of the MathScheme project [8] at McMaster University. The current principal goal of FFMM is to integrate and generalize computer algebra systems and automatic computer theorem proving systems. *Biform theories* is one of the key ideas in FFMM providing a formal context in which deduction and computation can be merged.

An axiomatic theory represents mathematical knowledge declaratively as a set of axioms while an algorithmic theory represents mathematical knowledge operationally as a set of algorithms. A biform theory is simultaneously an axiomatic theory and an algorithmic theory.

Briefly, a *biform theory* [1, Definition 2] in a general logic \mathbf{K} is a triple $T = (L, \Gamma, \Omega)$, where L is the language of \mathbf{K} represented by a set of symbols, Ω is a set of *transformers* for L and Γ is the set of axioms of T .

See Chapter 11 for an example of a biform theory.

The paper “Biform Theories in Chiron” [3] defines the notion of a biform theory, gives an overview of Chiron, and illustrates how biform theories can be formalized in Chiron.

2.4 Programming Language Choice

Objective Caml is the programming language we have chosen for implementing our MMS. It is a programming language which features strong static typing. If we use the language correctly, we can take advantage of static *type checking* to delegate the

verification of many of the invariants of the Chiron language to the programming language itself.

Furthermore, other features, such as exhaustiveness-checking for *pattern-matching* on variant types is very useful. Missing cases and redundant patterns will both produce a warning message to inform the program of this likely coding error.

Note that, when attempting to hide certain parts of the implementation by using a *private row type* (a recent feature of Objective Caml), we found that this did not interact very well with polymorphic variant types and sub-typing, and thus we had to abandon the use of this feature. Unfortunately that also meant giving up on completely hiding the details of our types.

Chapter 3

Goals

We present both the design goals and the implementation goals we would like to achieve at the first stage of the development of our mechanized mathematics system.

3.1 Design Goals

(1) Faithful embedding

To embed Chiron expression formation rules into the host programming language's type system as faithfully as possible.

(2) Abstract low-level details

To implement an API for bridging the low level Chiron data representation and the high level user input at the level of data-structures.

(3) Support for transformers and theories

To create name spaces for theories in *Chiron* to collect the transformers of theories.

(4) Support user-defined transformers

To provide tools for users to create their own transformers.

(5) Represent mathematics

To utilize biform theories in *Chiron* to represent mathematics.

(6) Support for running the Chiron transformers

To have a built-in meta-program for *reading* and *executing* operator applications of *Chiron*.

(7) Simplification

To have a special transformer to support basic simplifications.

(8) Beta reduction in *Chiron*

To implement a transformer corresponding to beta reduction for the application of a function abstraction.

3.2 Implementation Goals

(1) To hide the representation of the Chiron data-structures from the end user.

(2) To have the Chiron built-in operators pre-constructed, for the purposes of both code simplicity and run time efficiency.

(3) To design and implement the data structure for biform theories (and the name space environments) in *Chiron* carefully, so that the notion of biform theories can be translated precisely from its semantic definition to a code implementation.

-
- (4) To design the implementation for the notion of biform theories in a modular programming style.
 - (5) To improve the efficiency of the transformers in *Chiron* at the code level without affecting the API of the transformers.
 - (6) To support merging of name spaces, so that a large name space can be constructed from small name spaces.
 - (7) To support module inheritance for modules of biform theories, so that a new biform theory can be built up from certain parent biform theories.
 - (8) To create sample transformers which are built on top of the kernel theory.
 - (9) To have a well-designed testing facility for the development of the MMS.

3.3 Design Overview

We design and implement the type system of the MMS *Chiron* to embed Chiron expression formation rules into the host programming language's type system as faithfully as possible (Chapter 5 : **Type System**). The API for bridging the low level Chiron data representation and the high level user input is fully implemented in the modules of **Constructors** and **Builtin** (Chapter 6 : **Constructors**).

The system supports the creation of *name space* environments for theories in *Chiron* to collect the transformers of theories. Currently, **Basics** and **Logics**, are the two name spaces created for representing the kernel theories of *Chiron* (Chapter 7 : **Name Spaces**).

One special transformer, *simplify*, is built in our system to support basic simplifications (Chapter 9 : **Simplify**). Beta reduction is implemented as a transformer to utilize the transformers of the kernel theories of Chiron to represent the law of beta reduction in *Chiron* (Chapter 10 : **Beta Reduction**). The system represents Church numerals in the Chiron data-structure as a comprehensive testing for the implementation of beta reduction in *Chiron* (Chapter 12 : **Testing : Church Numerals in Chiron**).

A meta-tool, called *run*, is implemented for the system to find the transformers among Chiron expressions and execute them (Chapter 8 : **Run**). The creation of user-defined transformers is supported in our MMS (Chapter 13 : **User-defined Transformers**).

Chiron representation of biform theories is established in our MMS with experimental examples (Chapter 11 : **Biform Theories**).

3.4 Implementation Overview

The system is designed and implemented in a modular programming style (Chapter 4 : **Overview of Implementation**).

The code implementation for the type system of the MMS *Chiron* is hidden from the end user (Chapter 5 : **Type System**). The creation of Chiron expressions can be done through the constructor functions coded in the **Constructors** module. For the purposes of both code simplicity and run time efficiency, Chiron built-in operators are pre-constructed in the **Builtin** module (Chapter 6 : **Constructors**).

For every biform theory T in Chiron, the low level code implementation for the

transformers of T is implemented in a single module, named as *Implementation Module*. Inside the implementation module, we create submodules to implement the transformer algorithms in two layers, so that the run time efficiency of the algorithms gets improved without affecting their APIs. For the purposes of bundling together the various routines contained in the implementation module, we create a name space for the named transformers which correspond to these routines in T (Chapter 7 : **Name Spaces**).

The meta-tool *run* is implemented by separating the traversal process for finding the subexpressions of Chiron transformers from the input expression, and the execution process for running the named transformers (Chapter 8 : **Run**). Similarly, the transformer *simplify* is implemented by separating the traversal process for finding the subexpressions from the input expression which may require simplifications, and the code implementations for various simplification algorithm functions (Chapter 9 : **Simplify**).

The biform theory data structure in Chiron is designed in the **biform** module. It is implemented as collections of lists and hash tables (Chapter 11 : **Biform Theories**).

A well-designed testing facility for the development of the MMS is included in the current version of *Chiron*.

Chapter 4

Overview of Implementation

Chiron is designed in a modular programming style which is a powerful organizing principle for designing and implementing non-trivial programs. It breaks down the design of a program into individual components called *modules* which can be programmed and tested independently by grouping related sets of code together into a single module. Modular programming becomes a standard requirement for effective development and maintenance of programs and projects. This chapter will quickly go over the overall design of *Chiron* by introducing the modularization of the implementation for *Chiron*.

The two base modules, ***types*** and ***keywords***, build the fundamental type system of *Chiron*. Since the implementation of Chiron type system needs to be hidden from the rest of Chiron system and be hidden from end users, both the ***constructors*** module and the ***builtin*** module serve as interfaces between the low level implementation of the Chiron type system and the user. Users can only build expressions in *Chiron* by calling the constructor functions from the last two modules.











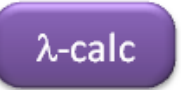




Type System	 	
Chiron Data Structure Construction	 	
System Services	 	
Name Spaces		
Foundation	Kernel Theories	Libraries
	 	 
Biform Data Structure		
Biform Theories	  	

Figure 4.1: Core Modules

messages and *printers* are two modules that provide system messages and printer services respectively.

- The *messages* module generates and collects all sorts of system related messages, such as errors and warnings, and stores them in one organized place.
- The *printers* module provides standard printing service for the system, and it should be able to generate output in L^AT_EX form in the future to give both the developer and the user more readable output.

The data structure called *biform* is declared for the notion of biform theories, which is the core idea of FFMM, a Formal Framework for Managing Mathematics [6]. The implementation for the representation of biform theories in *Chiron* takes two steps :

- (1) The first step is to define the transformer operators π in our MMS along with the implementation of the corresponding transformer algorithm functions $\hat{\pi}$. Again, the code implementations for the pairs of $(\pi, \hat{\pi})$ are coded in an independent module for each theory, such as the modules *basics*, *logicals*, *nats* and *lambdacalc*. In order to hide the low level implementation from the user, the *environment* data structure, named *name space*, acts as the interface for exporting the work done in those implementation level modules, so that the user can access the data of the transformer pair $(\pi, \hat{\pi})$ without seeing the low level implementation code.
- (2) The next step is to express the axioms of the language as one component of a biform theory. The axioms of the language are called *meaning formulas* in

Meta Tool	run
Special Transformer	simplify
Utility	compiler

Figure 4.2: Special Modules

Chiron. The modules *basics_bif*, *logicals_bif* and *nats_bif* are created by extending the work done in first step from the modules of *basics*, *logicals* and *nats*, with the addition of the corresponding *meaning formulas* for every pair of $(\pi, \hat{\pi})$ respectively.

Figure 4.2 shows three special modules which deserve to be introduced separately. The meta-program *run* is implemented in the module called **Run**. It does the actual work of finding the transformers among the expressions and applying the corresponding algorithms to get the instances of the transformers. A *simplify* gets implemented in the **Simplify** module at a very basic level only for boolean algebra and logic simplifications at this stage. Furthermore, a small *compiler* from the **Compiler** module is used for creating user defined transformers.

Lastly, testing is an important part for any system development. Figure 4.3 lists the modules related to the testing work. All sorts of testing routines have been

	Inputs	Suites
Tests Builders	test_input	test_suite_[x], x = 1,2, ...
Tests Design	testbox	
Tests Executors	test[x], x = 1,2, ...	
Tests Report	test_report	

Figure 4.3: Testing Modules

designed in the *testbox* module, while *test_input* and *test_suite_x*¹ are the modules used for preparing the actual testing data. Then, the *test[x]*² module calls the testing routines on the testing data to run the tests. A report is generated by the *test_report* module at the end.

¹*x* is replaced by a natural number ($x \geq 1$) at the code level, such as *test_suite_1* and *test_suite_2*.

²*x* is replaced by a natural number ($x \geq 1$) at the code level, such as *test1* and *test2*.

Chapter 5

Type System

5.1 Values & Expressions

As a derivative of NBG, Chiron is intended to be an enhanced version of STMM [7], a conservative extension from NBG under a preserving embedding. Although Chiron has a much richer syntax and more complex semantics than NBG, the models for Chiron, \mathbf{M}_{CHI} , contain exactly the same values as the models for NBG, \mathbf{M}_{NGB} . The formal specification of Chiron thus starts from the formulation of the fundamental mapping between the *values* in NBG set theory and the *expressions* in Chiron.

Definition 5.1.1 A *value* can be one of the following : [5, section 2.2]

- *set* : A set is a class which is also a member of a class.
- *class* : A class is an element of \mathbf{M}_{NGB} . A class is a collection of sets; each class is a collection of classes in NBG set theory. A class is *proper* if it is not a set.
- *superclass* : A superclass is a collection of classes, but need not be a class itself.

- *truth value* : A truth value is either \mathbf{T} representing *true* or \mathbf{F} representing *false*.
- *undefined value* : \perp is the value of undefined terms.
- *operation* : A mapping over superclasses, the truth values, and the undefined value.

The syntax of Chiron expressions is organized into four kinds of *expressions*, and every *expression* is a tree structure whose leaves are *symbols*. *Symbols* of Chiron will be introduced in the next section.

Definition 5.1.2 *Operators, types, terms, and formulas* are four special sorts of expressions.

Definition 5.1.3 An expression is *proper* if it is one of those four special kinds of expressions. An expression is *improper* if it is not proper.

Remark 5.1.4 *Proper expressions* denotes values; *improper expressions* do not denote anything.

The mapping from Chiron *expressions* to *values* is defined by Table 5.1 [5, section 2.3].

Expressions	denote	Values
Operators	\longrightarrow	Operations
Types	\longrightarrow	Superclasses
Terms	\longrightarrow	Classes [Sets, \perp]
Formulas	\longrightarrow	Truth values

Table 5.1: Values & Expressions

5.2 Expressions in Chiron

The two formation rules defined in [5, chap. 3] inductively define the notion of an *expression* of Chiron.

Expr-1 (Atomic expression)

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

Expr-2 (Compound expression)

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

where $n \geq 0$.

5.2.1 Data Type Choice for Chiron Expression

The *variant type* in OCaml is ideal for the purpose of representing the AST¹ of Chiron expressions. An atomic Chiron *expression* can be represented by a simple variant while a compound Chiron *expression* can be represented by a recursive variant. Recursive variant types are ideal for representing trees. Since Chiron expressions are categorized according to their first (leftmost) symbol, the constructor keywords defined in those variant type variables become the categorizing keywords for all Chiron expressions naturally.

¹AST - Abstract Syntax Tree, a finite, labeled, directed tree used in computer science.

Example 1

For a term variable x of type α represented in Chiron as

$$(\text{var}, x, \alpha),$$

we declare a variant type variable as

$$\text{variable} \rightarrow \text{Var of } \mathbf{symbol} * \mathbf{typ}$$

where **Var** is the constructor keyword for the variant type variable, and the variant *variable* contains two leaves which are the name of the *variable* [type **symbol** (Table 5.2)] and the type of the *variable* [type **typ** (Table 5.9)].

Finally, we use a technique called *tying the knot* for creating the Chiron types, as some of the types are not regular-recursive and we also wish to use *polymorphic variants* for subtyping.

Example 2

For a quotation term that denotes the construction that represents an expression e , it is represented in Chiron as

$$(\text{quote}, e),$$

we declare it as a subtype of the type **term** as

OCaml Code 1 Type declaration for type **quote** - types.ml

```

type 'e preterm =
  [
    .....
    | 'Quote of 'e
    | ..... ]

type 'e preproper = [ 'e preterm | ..... ]

type 'e preexpression = [ 'e preproper | ..... ]

type sexpression = sexpression preexpression
and proper       = sexpression preproper

type term        = sexpression preterm

```

Notice that, in the Chiron type system, since type **term** is a subtype of type **proper** and type **proper** is a subtype of type **sexpression**, type **sexpression** can only be declared if type **proper** has already been created and type **proper** can only be declared if type **term** has been already created. Therefore, the type declaration for type **term** appears before the type declaration for type **sexpression** at the code level, and we need to leave a “hole” for **sexpression**.

Without using a free polymorphic variable **'e**, type **sexpression**, the (eventual) type of the value e in a quotation term, cannot be part of the type declaration for type **quote** which is a subtype of type **term** since type **sexpression** has not been created yet. By leaving this open, we declare the type **term** as a parametric (polymorphic) variant type. Later we “tie the knot” by instantiating **'e** to type **sexpression** after the type **sexpression** is created. The type declaration of type **quote** can just use that parametric value **'e** to represent the type of the value e , namely type **sexpression** in

our Chiron type system (OCaml Code 1).

5.2.2 Symbol

Since an *expression* is a tree whose leaves are symbols, type **symbol** is the first data type we create as it will be used heavily in all other type declarations in the type system of *Chiron*.

The set of symbols, \mathcal{S} , is neither fixed nor well-ordered, but countably *infinite*. They are used to classify expressions, identify different categories of expressions, and name the built-in operators. The implementation of *Chiron* declares type **symbol** with two constructors (Table 5.2).

constructors	values
K	keyword
S	string

Table 5.2: type **symbol**

The first constructor for type **symbol** requires a value of type **keyword** as part of its construction, and is used to designate particular *distinguished symbols*, namely the keywords of the language. Type **keyword** is another abstract type we declare in *Chiron*. Table 5.3 lists all the keywords that are included in the current version of *Chiron*. This table is an extension of the table [5, table 1].

op	type	formula	op-app	var
type-app	dep-fun-type	fun-app	fun-abs	if
exist	def-des	indef-des	quote	eval
set	class	expr	expr-sym	expr-op
expr-type	expr-term	expr-term-type	expr-formula	in
type-equal	term-equal	formula-equal	not	or
all	teval	feval	true	false
empty_set	undefined	and	implies	defined_in
quasi_equal	uint			

Table 5.3: The **keyword** of Chiron.

Type **keyword** is assembled in a separate module in *Chiron* for two reasons. Firstly, since all those symbols will be used heavily for constructing *Chiron* expressions as the base elements, a special pre-defined type which collects them all can ensure all leaves of *Chiron* expressions with these symbols are type safe easily, and then they can also be extended or modified in one place without having to find and change them everywhere.

The second constructor for type **symbol** requires a value of a user **string** as part of its construction, so that the size of the Chiron symbols is unbounded by arbitrary string values.

5.2.3 S-Expression

Definition 5.2.1 An expression is a S-expression (with commas in places of spaces)² that exhibits the structure of a tree whose leaves are symbols $\in \mathcal{S}$.

Since the syntax of constructing³ a Chiron *expression* is extensively employed in

²The representation of a *S-expression* in Lisp is written with its elements separated by whitespace; while an *expression* in Chiron paper is written with its elements separated by commas.

³A construction is a set that represents the syntactic structure of an expression.

the Lisp family of programming languages, the *expression* in Chiron adopts the name which is the name used for the expression in Lisp, namely *S-expressions*. Chiron *S-expressions* also adopts the convention of using prefix notation from Lisp, the leftmost symbol of the tree for a Chiron *S-expression* categorizes the type of the expression.

We declare the type **sexpression** for *S-expressions* as an enumerated variant type shown in Table 5.4 :

type	subtype
sexpression	proper unknown

Table 5.4: type **sexpression**

i.e., an S-expression [type **sexpression**] is either a proper expression [type **proper**], or an improper expression [type **unknown**]. The following two sections introduce the notion of proper expressions and improper expressions in terms of their implementation in *Chiron*.

5.2.4 Proper S-Expression

The set of 13 formation rules below, from [5, chap. 3] defines the notion of a *proper S-expression* in Chiron.

P-Expr-1 (Operator)

$$\frac{s \in \mathcal{S}, \mathbf{kind}[k_1], \dots, \mathbf{kind}[k_{n+1}]}{\mathbf{operator}[(\text{op}, s, k_1, \dots, k_{n+1})]}$$

where $n \geq 0$.

P-Expr-2 (Operator application)

$$\frac{\text{operator}[(\text{op}, s, k_1, \dots, k_{n+1})], \text{expr}[e_1], \dots, \text{expr}[e_n]}{\text{p-expr}[(\text{op-app}, (\text{op}, s, k_1, \dots, k_{n+1}), e_1, \dots, e_n) : k_{n+1}]}$$

where $n \geq 0$ and $(k_i = \text{type}$ and $\text{type}[e_i])$, $(\text{type}[k_i]$ and $\text{term}[e_i])$, or $(k_i = \text{formula}$ and $\text{formula}[e_i])$ for all i with $1 \leq i \leq n$.

P-Expr-3 (Variable)

$$\frac{x \in \mathcal{S}, \text{type}[\alpha]}{\text{term}[(\text{var}, x, \alpha) : \alpha]}$$

P-Expr-4 (Type application)

$$\frac{\text{type}[\alpha], \text{term}[a]}{\text{type}[(\text{type-app}, \alpha, a)]}$$

P-Expr-5 (Dependent function type)

$$\frac{\text{term}[(\text{var}, x, \alpha)], \text{type}[\beta]}{\text{type}[(\text{dep-fun-type}, (\text{var}, x, \alpha), \beta)]}$$

P-Expr-6 (Function application)

$$\frac{\text{term}[f : \alpha], \text{term}[a]}{\text{term}[(\text{fun-app}, f, a) : (\text{type-app}, \alpha, a)]}$$

P-Expr-7 (Function abstraction)

$$\frac{\text{term}[(\text{var}, x, \alpha)], \text{term}[b : \beta]}{\text{term}[(\text{fun-abs}, (\text{var}, x, \alpha), b) : (\text{dep-fun-type}, (\text{var}, x, \alpha), \beta)]}$$

P-Expr-8 (Conditional term)

$$\frac{\text{formula}[A], \text{term}[b : \beta], \text{term}[c : \gamma]}{\text{term}[(\text{if}, A, b, c) : \delta]}$$

$$\text{where } \delta = \begin{cases} \beta & \text{if } \beta = \gamma \\ (\text{op-app}, (\text{op}, \text{class}, \text{type})) & \text{otherwise} \end{cases}$$

P-Expr-9 (Existential quantification)

$$\frac{\text{term}[(\text{var}, x, \alpha)], \text{formula}[B]}{\text{formula}[(\text{exist}, (\text{var}, x, \alpha), B)]}$$

P-Expr-10 (Definite description)

$$\frac{\text{term}[(\text{var}, x, \alpha)], \text{formula}[B]}{\text{term}[(\text{def-des}, (\text{var}, x, \alpha), B) : \alpha]}$$

P-Expr-11 (Indefinite description)

$$\frac{\text{term}[(\text{var}, x, \alpha)], \text{formula}[B]}{\text{term}[(\text{indef-des}, (\text{var}, x, \alpha), B) : \alpha]}$$

P-Expr-12 (Quotation)

$$\frac{\text{expr}[e]}{\text{term}[(\text{quote}, e) : (\text{op-app}, (\text{op}, \text{expr}, \text{type}))]}$$

P-Expr-13 (Evaluation)

$$\frac{\text{term}[a], \text{kind}[k]}{\text{p-expr}[(\text{eval}, a, k) : k]}$$

Proper expressions can be categorized into four different kinds according to the mapping relations shown in Table 5.1. *Chiron* creates four variant types [**operator**, **typ**, **term** and **formula**] to represent the four sorts of proper expressions [*operator*, *type*, *term* and *formula*] respectively. Table 5.5 extends Table 5.1 to include the meaning of each mapping in terms of Chiron type system.

Expressions & Values	Chiron Type System
Operators \Rightarrow Operations	A proper expression to which the type operator is assigned
Types \Rightarrow Superclasses	A proper expression to which the type typ is assigned
Terms \Rightarrow Classes [Sets, \perp]	A proper expression to which the type term is assigned
Formulas \Rightarrow Truth values	A proper expression to which the type formula is assigned

Table 5.5: **Proper** Expressions in Chiron

Proposition 1 *The formation rules assign a unique expression to each proper expression.* [5, proposition 3.1]

Namely, a *proper expression* in *Chiron* will be one of types of **operator**, **typ**, **term** and **formula** with no exception. (Table 5.6)

type	subtype
proper	operator
	typ
	term
	formula

Table 5.6: type **proper**

Most of the type declarations for the four base types, **operator**, **typ**, **term** and **formula** are created based on the formation rules straightforwardly, while some of them are created differently. Details will be discussed in the following three sections.

Type Declaration - Part 1

For a simple case, such as

P-Expr-3 (Variable)

$$\frac{x \in \mathcal{S}, \text{type}[\alpha]}{\text{term}[(\text{var}, x, \alpha) : \alpha]}$$

$\text{term}[\dots : \cdot]$ shows that a variable in Chiron is a subtype of type **term**. Hence, we create type **TermVar** as the subtype of type **term** to represent variables. $[(\text{var}, x, \alpha) : \alpha]$ shows that a variable needs two pieces of data as its components, x and α , where x is a **symbol** for representing the name of the variable ($x \in \mathcal{S}$) and α is a Chiron **typ** for representing the type of the variable ($\text{type}[\alpha]$). As a result, we have the type declaration for Chiron variables as **TermVar** of **symbol** * **typ**.

The above approach can be used for creating a series of Chiron types based on the set of 13 formation rules with the exceptions for **P-Expr-1**, **P-Expr-2** and **P-Expr-13**. We need to introduce type **kind** and type **kinded** before creating the Chiron types which correspond to those three formation rules.

Type Declaration - Part 2

type **kind**

Kinds are the expressions assigned to types, terms and formulas. A proper expression e is said to be an expression of **kind** k if [5, chap. 3]

- (1) $k = \text{type}$ and e is a type, or

- (2) $\text{type}[k]$ and e is a term of type k , or
- (3) $k = \text{formula}$ and e is a formula.

Therefore, the type declaration for type **kind** is composed of three constructors to reflect the three cases shown in the definition above. (Table 5.7)

constructors	values
KType	
KFormula	
Kind	typ

Table 5.7: type **kind**

Instead of using type **typ** as part of the type declaration for representing an expression of **kind** type, we use a type value **KType** to indicate that an expression is a type in general. A proper expression e is said to be an expression of **kind** type as long as e is a type, the exact type of e is irrelevant. Similarly, **KFormula** is created for representing an expression of **kind** formula if the expression is a formula. However, if the expression is a term of type k , the type declaration for representing an expression of $\text{type}[k]$ needs to include the exact type k as part of its declaration. We use a type value **Kind** to start the type declaration for representing **kind** $\text{type}[k]$ follow by a **typ** value to include the exact type k .

type **kinded**

Type **kinded** is created in addition to type **kind** to include the expression that is categorized by type **kind** as part of its declaration. The type declaration for type **kinded** is prepared by adding one component to each type declaration shown in Table 5.7 for storing the expression it points to. (Table 5.8)

constructors	values
KDType	type
KDFormula	formula
KDTerm	term * typ

Table 5.8: type **kinded**

Type Declaration - Part 3

By having both type **kind** and type **kinded** be declared, we create the Chiron types which correspond to the formation rule **P-Expr-1**, **P-Expr-2** and **P-Expr-13** as the following.

P-Expr-1 (Operator)

$$\frac{s \in \mathcal{S}, \mathbf{kind}[k_1], \dots, \mathbf{kind}[k_{n+1}]}{\mathbf{operator}[(\text{op}, s, k_1, \dots, k_{n+1})]}$$

where $n \geq 0$.

An operator is composed of a **symbol** for representing the name of the operator and $n + 1$ **kinds** for representing the **kind** for both the inputs (n inputs) and the output (1 output). Since n is unknown, we use *list* data structure to represent the collection of the $n + 1$ **kinds**. Instead of using a single list to collect all the $n + 1$ **kinds**, we declare the type **operator** to have one list for collecting the first n **kinds** and one single component to include the last **kind**. Thus, the type declaration for Chiron operators is **Operator of symbol * kind list * kind**.

P-Expr-2 (Operator application)

$$\frac{\mathbf{operator}[(\mathbf{op}, s, k_1, \dots, k_{n+1})], \mathbf{expr}[e_1], \dots, \mathbf{expr}[e_n]}{\mathbf{p-expr}[(\mathbf{op-app}, (\mathbf{op}, s, k_1, \dots, k_{n+1}), e_1, \dots, e_n) : k_{n+1}]}$$

where $n \geq 0$ and ($k_i = \mathbf{type}$ and $\mathbf{type}[e_i]$), ($\mathbf{type}[k_i]$ and $\mathbf{term}[e_i]$), or ($k_i = \mathbf{formula}$ and $\mathbf{formula}[e_i]$) for all i with $1 \leq i \leq n$.

Since $\mathbf{p-expr}[\dots : \cdot]$ does not tell which base type an operator application should belong to, an expression of an operator application can be a type, a term of type α , or a formula. We create three Chiron types, one for each case, namely

- TypeApplyTerm of **symbol** * **kinded** list
- ApplyTerm of **symbol** * **kinded** list * **typ**
- FApplyTerm of **symbol** * **kinded** list

The **symbol** occurring in those type declarations is used for representing the name of the operator application. We use a *list* of type **kinded** to store the input expressions of the operator application, so that, not only the input expressions, but also the types of the input expressions can be extracted directly under such declaration. In particular, a **typ** value is needed as part of the type declaration for a **term** type operator application to represent the type of the return value.

P-Expr-13 (Evaluation)

$$\frac{\mathbf{term}[a], \mathbf{kind}[k]}{\mathbf{p-expr}[(\mathbf{eval}, a, k) : k]}$$

Similar to the approach used for **P-Expr-2**, we create three Chiron types to map

formation rule **P-Expr-13** to the Chiron type system. Since the expression e involved in the type **P-Expr-13 (Evaluation)** denotes the construction that represents an expression, the type declaration for an evaluation takes the expression e directly as part of its declaration to keep the construction.

- TEval of **sexpression**
- Eval of **sexpression * typ**
- FEval of **sexpression**

The type declarations of those four base types, **operator**, **typ**, **term** and **formula**, are shown in table 5.9, table 5.10, table 5.11 and table 5.12, respectively.

constructors	values	formation rule
TypeBase	symbol	
TypeApplyTerm	symbol * kind list	P-Expr-2 (type typ Operator application)
TypeApp	typ * term	P-Expr-4 (Type application)
TypeDepFun	symbol * typ * typ	P-Expr-5 (Dependent function type)
TEval	sexpression	P-Expr-13 (typ Evaluation)

Table 5.9: type **typ**

constructors	values	formation rule
Operator	symbol * kind list * kind	P-Expr-1 (Operator)

Table 5.10: type **operator**

constructors	values	formation rule
FBase	symbol	
FApplyTerm	symbol * kinded list	P-Expr-2 (type formula Operator application)
FExists	symbol * typ * formula	P-Expr-9 (Existential quantification)
FAll	symbol * typ * formula	P-Expr-9 (Universal quantification) extension
FEval	sexpression	P-Expr-13 (formula Evaluation)

Table 5.11: type **formula**

constructors	values	formation rule
TermBase	symbol	
ApplyTerm	symbol * kinded list * typ	P-Expr-2 (type term Operator application)
TermVar	symbol * typ	P-Expr-3 (Variable)
FunApp	term * term	P-Expr-6 (Function application)
FunAbs	symbol * typ * term	P-Expr-7 (Function application)
IfTerm	formula * term * term	P-Expr-8 (Conditional term)
DefDescr	symbol * typ * formula	P-Expr-10 (Definite description)
IndefDescr	symbol * typ * formula	P-Expr-11 (Indefinite description)
Quote	sexpression	P-Expr-12 (Quotation)
Eval	sexpression * typ	P-Expr-13 (term Evaluation)
Construction	unknown	unknown term Construction

Table 5.12: type **term**

5.2.5 Improper S-Expression

An expression is *improper* if it is *NOT* one of the four special kinds of expressions. We declare the type **unknown** as shown in Table 5.13 to represent improper expressions.

constructors	values
US	symbol
UE	sexpression list
UInt	int
UUniv	Universal Type

Table 5.13: type **unknown**

There are four subtypes under the type **unknown** in the current version of *Chiron*. The first two are officially defined in the Chiron paper [5] for defining the notion of an *expression* of Chiron while the later two are experimental, being investigated currently.

(1) US of **symbol**

This is created based on the formation rule **Expr-1 (Atomic expression)** defined in Chiron paper [5].

Expr-1 (Atomic expression)

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

An atomic expression which only contains a **symbol** value is an improper *expression* of Chiron.

(2) UE of **sexpression** list

Again, this is created based on the formation rule **Expr-2 (Compound expression)** defined in Chiron paper [5].

Expr-2 (Compound expression)

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

where $n \geq 0$.

A compound expression that is a collection of Chiron *expressions* is an improper *expression* of Chiron.

(3) UInt of **int**

Basically, this experimental type is used particular for representing integers in *Chiron* for the **Nats** library. For any integer n , the Chiron data structure of the form $(U\text{Int}, n)$ is used in *Chiron* as its Chiron representation. The `int` library used in *Chiron*'s implementation directly uses the built-in `Int32` module from OCaml built-in libraries.

(4) UUniv of **t**, where **t** is a universal type.

Universal type will be supported in the future to make the type system of *Chiron* flexible and extensible. *Chiron* should be able to extend its built-in type system either by the developers who can design their own type modules whenever convenience for their mathematical needs, or by users who can easily load any particular type module any time they want from a pre-coded universal type library provided by the *Chiron* system, but not pre-loaded for a standard bootup of the system.

5.2.6 quoted Term

Type Declaration

Type **quoted**, a special type for quoted terms, is defined as in Table 5.14 :

constructors	values
Quote	sexpression

Table 5.14: type **quoted**

The constructor *Quote*, which has one argument value of type **sexpression**, is also

one of the constructors that is used in the type declarations for the subtypes of type **term**; a S-expression of type **quoted** is essentially a subtype of type **term**. Namely, the coercion, (**quoted** \rightarrow **term**), is valid in *Chiron*.

Purpose

Type **quoted** is created particularly both for

- (1) Code *simplicity* : to avoid all kinds of inefficient code used mainly for type checking and coercions required all over the system for the purpose of appeasing the OCaml compiler.
- (2) Concept *clarity* : to make quoted **terms** stand out from all other sorts of **terms**.

For fairly large amounts of code in the current version of *Chiron*, the implementation of transformer functions needs to ensure that both the inputs and the output should be quoted terms. Without the creation of type **quoted**, the signature of all the transformer functions appear to just take **term** type inputs and return **term** type output generally, and then, *Chiron* needs extra efforts both for the input checking to ensure that all the arguments are properly quoted and the output coercion to ensure that the output value should again be quoted strictly. For instance,

Example 3 f is a transformer that takes one **input** and returns one **output**.

Pseudocode

f (input : **term**) =

if (input is a term which is properly quoted) **then**

```
do something for the expression inside the quotation, say e.  
get the instance of e, e_inst, according to the semantic of the transformer, f.  
return output, which is [e_inst].  
else  
do nothing, return as it is
```

By having the existence of the special type **quoted**, the signature of all those transformer functions could specify the types for both the inputs and the output to be quoted terms precisely. The code shown in the previous example would be simplified as

Example 4 *f* is a transformer that takes one **input** and returns one **output**.

Pseudocode

```
f (input : quoted) =  
do something for the expression inside the quotation, say e.  
get the instance of e, e_inst, according to the semantic of the transformer, f.  
return output, which is [e_inst].
```

Thanks to the type checking provided by the OCaml compiler, the type validation on the inputs will be checked at compilation time automatically. As long as the code for constructing the Chiron expressions is correct by passing through the OCaml

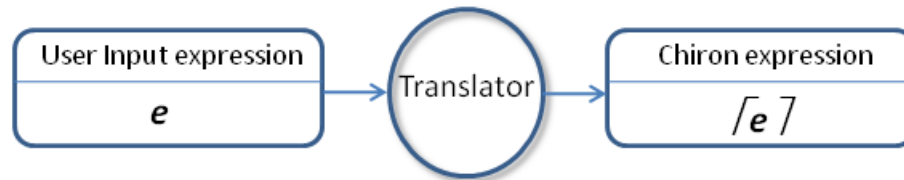


Figure 5.1: Translator

compiler without errors, we ensure the correctness on the type of the inputs for all those transformers for free.

When to Quote

When users are writing their expressions at the front end of the MMS, a *translator* for translating user inputs from the GUI to Chiron recognizable expressions, will quote all the arguments for any expression of an operator application automatically. Because of that, when *Chiron runs*⁴ on the resulting expressions, it is able to apply properly the transformers which correspond to those operator applications with *quoted terms* as their arguments.

Question :

Why does the job of appending those quotations ($\lceil \ \rceil$) need to be done during the translation step? Is it reasonable to do it during the *run* procedure instead?

⁴*Run* is a meta-program in the MMS for actually running the transformer which correspond to an operator application for Chiron expressions. See Chapter 8 : **Run** for details.

Answer :

From Figure 5.1, we can easily see that there is no quotation around any expression, e , which is entered by the user from some form of GUI to be developed in the near future. The meaning of a quotation should definitely be understood below the end user level. We should not expect end users to routinely require special annotations for quotation around their input to indicate that the input is to be understood (by the system) first as a syntactic expression.

On the other hand, all expressions which are assumed to be valid in *Chiron* need to be properly quoted. This is an assumption made by Chiron. Therefore, whenever *Chiron* detects an operator application, quotations need to be there as the prerequisite for all expressions involved in the operator application.

As a result, the job of appending these quotations to the expressions at the user level needs to be done in between the GUI and our MMS automatically and silently; that is where the *translator* belongs. The *translator* finds all expressions that are parts of some operator application, and puts the quotations around the expressions.

Now, it is easy to answer the second half of the question. Since *run* is a *Chiron* internal meta-program, it assumes that all valid expressions are properly quoted. Therefore, it is more useful to have the translator at the user-interface level than at the “run” layer.

Chapter 6

Constructors

The implementation details of the embedding of Chiron terms as OCaml terms should be hidden from the end user to prevent all kinds of unexpected harm to the system. The system should build a black box for bridging the gap from the underlying Chiron type system to the actual syntax representation of Chiron expressions. The module ***Constructors*** is created for this purpose. Whenever an arbitrary type of Chiron expression needs to be constructed, a corresponding function call for expression of that type from the ***Constructor*** module should be available to construct the expression in the Chiron syntax; all the necessary type checkings for every element of the resulting expression will be done by the constructor functions.

6.1 Chiron Types

Constructors for the type system of *Chiron* can be prepared simply by creating functions of the following form :

Suppose **t** is an elementary type in *Chiron*, defined as follows :

constructors	values
contr_of_t ₁	$v_{11}, v_{12}, \dots, v_{1n}$
contr_of_t ₂	$v_{21}, v_{22}, \dots, v_{2n}$
⋮	⋮
contr_of_t _m	$v_{m1}, v_{m2}, \dots, v_{mn}$

Table 6.1: type **t**

Then, the set of constructor functions for type **t** will be coded as :

constructors	values	constructor functions
contr_of_t ₁	$v_{11}, v_{12}, \dots, v_{1n}$	$f(v_{11}, v_{12}, \dots, v_{1n}) = \text{contr_of_t}_1(v_{11}, v_{12}, \dots, v_{1n})$
contr_of_t ₂	$v_{21}, v_{22}, \dots, v_{2n}$	$f(v_{21}, v_{22}, \dots, v_{2n}) = \text{contr_of_t}_2(v_{21}, v_{22}, \dots, v_{2n})$
⋮	⋮	⋮
contr_of_t _m	$v_{m1}, v_{m2}, \dots, v_{mn}$	$f(v_{m1}, v_{m2}, \dots, v_{mn}) = \text{contr_of_t}_m(v_{m1}, v_{m2}, \dots, v_{mn})$

Table 6.2: Constructors for type **t**

Next, we use the type **formula** as an example to show the creation of the constructors for Chiron types. This is declared in *Chiron* as the following :

OCaml Code 2 type **formula** - type.ml

```

type 'e formula =
  [ 'FBase of symbol
  | 'FApplyTerm of symbol * 'e prekinded list
  | 'FExists of symbol * 'e pretyp * 'e preformula
  | 'FAll of symbol * 'e pretyp * 'e preformula
  | 'FEval of 'e ]

```

Therefore, the set of constructors for the type **formula** is given by :

OCaml Code 3 constructors for **formula** - types.ml

```

let form_base (s:symbol) = `FBase s

let apply_f ('Operator (s,kl,k):operator) (el: sexpression list) : formula =
  if k <> KFormula then
    failwith (Printf.sprintf "Applying a non-formula operator (%s)
                          and expecting a formula" (to_string s))
  else if List.length kl <> List.length el then
    failwith "Applying operator to expression with different lengths"
  else
    `FApplyTerm (s, List.map2 kinding el kl)

let exists (x:symbol) (t:typ) (b:formula) = `FExists (x,t,b)

let fall (x:symbol) (t:typ) (b:formula) = `FAll (x,t,b)

let eval_form (e:sexpression) = `FEval e

```

For every subtype of type **formula**, a constructor function is created to build the corresponding Chiron expression. For all the subtypes in the example of type **formula** given, except the subtype **FApplyTerm**, the arguments of the constructor functions reflect exactly the components of the type definitions. For instance, the subtype **FAll** contains three components in its type definition, namely a **symbol**, a **typ** and a **formula**. Accordingly, the constructor function for constructing the **FAll** Chiron expression takes three arguments, namely a **symbol**, a **typ** and a **formula**. For the subtype **FApplyTerm**, the first argument of its constructor takes an **operator** instead of a **symbol**, and the **symbol** can be extracted directly from the **operator** argument by pattern matching.

The above programming approach used for creating constructor functions based on the type system of *Chiron* is bijective and total. On the other hand, the type system of

Chiron is founded on the 13 formation rules for proper expressions, which are defined in [5, chap. 3], along with the addition 4 formation rules for improper expressions given in the previous chapter, *Type System*; the mapping between the Chiron type system and those formation rules, is bijective as well. Furthermore, by proposition 3.1 from the Chiron paper [5], the 13 formation rules assign a unique proper expression to each proper expression. That is, the resulting Chiron expression, constructed through the **Constructors** module, for any unique input proper expression is unique.

6.2 *builtin* Module

The ***builtin*** module is created specially for defining

- (1) Useful constructors for constructing elementary values [\perp , T, F, **empty**] used in *Chiron*.

Special values such as \perp , T, F, and **empty**, are expected to be used everywhere in the *Chiron* system extremely often. Having them constructed as built-in *Chiron* expressions during the initialization bootup process, can reduce both the compilation time and the running time of *Chiron*, since the low level constructions of those base expressions just need to be done once inside the ***builtin*** module during the compilation time.

Table 6.3 lists the special values constructed in the current version of *Chiron* and the corresponding variable names [types.ml] used at the OCaml implementation level.

values	variable names	type
\perp	undefined	term
empty	empty_set	term
T	truef true_exp	term sexpression
F	falsef false_exp	term sexpression

Table 6.3: Special Values

- (2) Built-in operators defined in Table 2 of [5].

Table 6.4 lists the corresponding variable names [types.ml] coded in *Chiron* for all those built-in operators.

- (3) Operators in addition to the 15 pre-defined built-in operators. Table 6.5 lists the corresponding variable names [types.ml] coded in *Chiron* for all these extra operators.

Built-In Operator	Variable Name in OCaml
(op, formula-and, formula, formula, formula)	formula_and
(op, formula-implies, formula, formula, formula)	formula_implies
(op, defined-in, (op-app, (op, class, type)), type, formula)	defined_in
(op, quasi-equal, (op-app, (op, class, type)), (op-app, (op, class, type)), type, formula)	quasi_equal
(op, is-empty-set, (op-app, (op, class, type)), formula)	is_empty_set

Table 6.5: Additional Built-In Operators in Chiron

Built-In Operator	Variable Name in OCaml
(op, set, type)	set
(op, class, type)	clas
(op, expr, type)	expr
(op, expr-sym, type)	expr_sym
(op, expr-op, type)	expr_op
(op, expr-type, type)	expr_type
(op, expr-term, type)	expr_term
(op, expr-term-type, (op-app, (op, expr-type, type)), type)	expr_term_typ
(op, expr-formula, type)	expr_formula
(op, in, (op-app, (op, set, type)), (op-app, (op, class, type)), formula)	in_op
(op, type-equal, type, type, formula)	type_equal
(op, term-equal, (op-app, (op, class, type)), (op-app, (op, class, type)), type, formula)	term_equal
(op, formula-equal, formula, formula, formula)	formula_equal
(op, not, formula, formula)	formula_not
(op, or, formula, formula, formula)	formula_or

Table 6.4: Built-In Operators in Chiron

6.3 Constructors for Meaning Formulas

The language L , which is one of the components in a biform theory¹ $T = (L, \Gamma, \Omega)$, is a set of symbols. Each symbol is either the name of a concept of T or the name of a transformer of T . (Ω is the set of transformers of T .) Both the concepts and the transformers of T are represented as **operators** in *Chiron*.

Γ is the set of axioms of T . For every operator in T , there are one or more axioms to specify the meaning of the operator. Those axioms are also called the *meaning*

¹For a complete definition, see Chapter 11 : Biform Theories.

formulas of T .

The meaning formulas firstly specify the type of the inputs required for the operator, and then express the meaning of the operator in the form of $A \equiv B$, where A is used to represent the expression of the operator application and B explains the actual meaning of the operator.

A simple example is :

$$\begin{aligned} \forall e : E . (\text{is-p-expr} :: E, \text{formula})(e) &\equiv \\ e \downarrow E_{\text{op}} \vee e \downarrow E_{\text{ty}} \vee e \downarrow E_{\text{te}} \vee e \downarrow E_{\text{fo}}. \end{aligned}$$

The meaning formula first specifies the type of the input as E , namely a Chiron expression. The formula “ $(\text{is-p-expr} :: E, \text{formula})(e)$ ” on the left hand side of the formula equality represents the expression of an operator application for the operator is-p-expr . The right hand side “ $e \downarrow E_{\text{op}} \vee e \downarrow E_{\text{ty}} \vee e \downarrow E_{\text{te}} \vee e \downarrow E_{\text{fo}}$ ” explains the actual meaning of the is-p-expr operator application, namely a Chiron expression is a proper expression if and only if it is an **operator** type Chiron expression, or a **typ** type chiron expression, or a **term** type Chiron expression, or a **formula** type Chiron expression.

For operators in the kernel theories of Chiron, the constructors of meaning formulas are prepared in the **Constructors** module; and for each (non-kernel) operator in *Chiron*, the meaning formula constructor is currently prepared in the same module where the operator is created. Examples for both cases will be shown in Chapter 7 and Chapter 11.

Chapter 7

Name Spaces

A `name space` in *Chiron* is used for bundling together the low level implementations for the named transformers for each biform theory. This chapter presents the implementation details for Chiron transformers, followed by a complete introduction to the notion of `name spaces`.

7.1 Transformers in Chiron

Deduction and computation rules are represented in Chiron as algorithms called `transformers`, which are intended to be expressions transforming algorithms that preserve or modify meaning in a prescribed way. A transformer could be an evaluator, a rewrite rule, a rule of inference, a decision procedure, a simplifier, a translation from one language to another, etc.

In Chiron, let L be a language of Chiron. An *n-ary transformer* Π in L is a pair $(\pi, \hat{\pi})$, where

- π is an *n-ary operator* $(s :: E, \dots, E)$ in L (with E occurring $n + 1$ times), and

s is a **symbol** used to represent the name of the operator.

- $\hat{\pi}$ is the corresponding *algorithm* of the transformer.

(1) Operators

Operators denote operations. Basically, the definition of an operator specifies the name of the transformer and the signature of the transformer function. An operator in Chiron is not meaningful for evaluation unless it is applied with proper inputs; an operator application can be evaluated by the meta-program *run* which will be discussed in the next chapter.

(2) Algorithm

An algorithm implements a (possibly partial) function $f_{\hat{\pi}} : \varepsilon^n \rightarrow \varepsilon$, where ε is a set of syntactic entities, the *expressions*.

As the system keeps growing, not only could the complexity of the relations between all the operators and algorithms cause difficulty, but also the size of system code could lower the system maintainability. To have all the transformers implemented in a single module is definitely a bad system design decision. Instead, we create different modules for different theories.

In general, there are two types of modules for storing the implementation of all sorts of transformers :

- **Kernel Theory Modules**

Built-in operators for Chiron, which are loaded automatically at the startup of the MMS.

- **Library Modules**

Non-built-in operators, defined on top of the built-in operators from kernel theories. In the future, libraries can be loaded at the request of users after the system is booted.

For each kernel theory module or library module, *Chiron* collects and organizes the code level implementation for the operators and the algorithm function routines of the transformers corresponding to the operators into a *name space* which is implemented by using a special data structure, called an *environment*. It acts as the interface for exporting the transformers, which are implemented inside the module, to the rest of the system in a clean and organized way. Because of that, we also call a kernel theory module or a library module an *Implementation Module*.

7.2 Implementation Modules

In the current version of *Chiron*, there are two kernel theory modules, called **Basics** and **Logicals**. In addition, there are two libraries. One library, called **lambdacalc**, currently contains only one transformer, which is a simple version of beta reduction; the other library, called **Nats**, is an experimental module for the theory of the natural numbers, implemented at a very basic level.

7.2.1 Module Structure

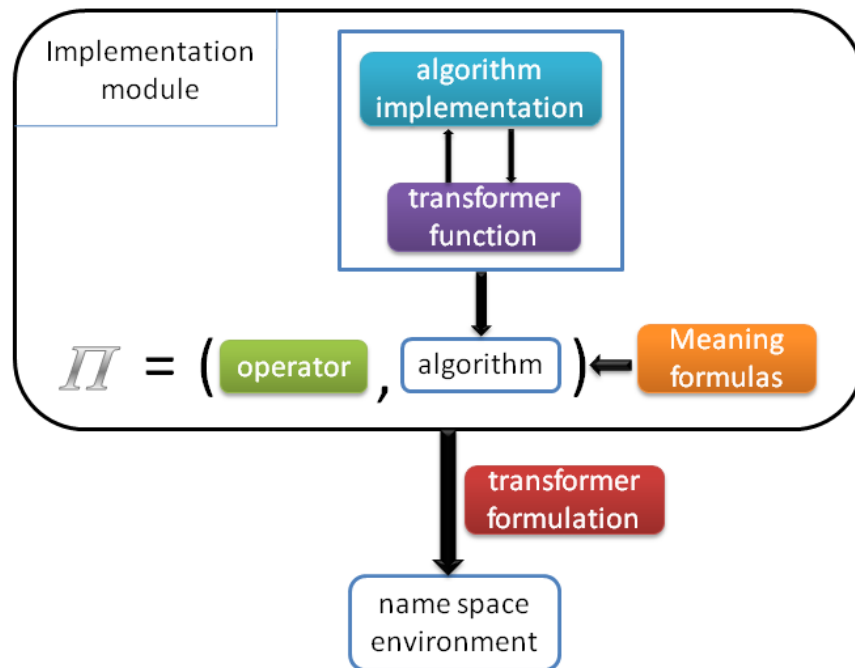


Figure 7.1: Module Structure

The implementation structure for building a kernel theory or a library contains five parts :

(1) **Operator definition**

Creates the operator S-expression, E_{op} , based on the semantic definition of an operator.

(2) **Constructors for meaning formulas**

Creates the constructor function used for expressing the meaning formulas of the transformer.

(3) **Transformer function**

Creates the function call for the transformer algorithm $\hat{\pi}$.

$\hat{\pi}$ firstly checks to see if all the inputs of the transformer function are in type **quoted**. i.e. a S-expression e needs to be properly quoted so that it denotes the syntactic meaning of e and is thus always defined. If any of the input expressions fails the “quotation” checking, the transformer function basically does nothing and simply returns the input expression as its output. In the case of success, $\hat{\pi}$ calls a proper algorithm routine written in the algorithm implementation submodule of the current module to generate the output of the $\hat{\pi}$.

(4) **Algorithm implementation**

Creates the function call in a submodule of the main module for the algorithm.

The implementation written in this submodule can concentrate mainly on the efficiency of the implementation code without worrying about all sorts of type coercions required as a regular Chiron function.

(5) **Transformer formulation**

Creates the Chiron transformer pair $(\pi, \hat{\pi})$.

Each pair of $(\pi, \hat{\pi})$ represents one Chiron transformer, which is one of the two components of a *rule* in a biform theory; the second component of a Chiron *rule* is a rule-less formula, called the *meaning formula*. The constructions of *meaning formulas* will be presented in Chapter 11.

7.2.2 π in *Chiron*

Notice that there is one minor difference between the formal definition of a π and the π here at the implementation level of a theory in *Chiron*. The π defined in the formal specification of Chiron is a **symbol** for representing the name of the operator, the π here, however, is an **operator**

$$O = (\text{op}, s, k_1, \dots, k_{n+1})$$

which in fact contains more information than just the name of the operator, since the **symbol** s is the part of the operator O that serves as the name of the operator.

The reason for this modification at this level of implementation is that the name space environment for exporting the implementation of the theory module needs to include all the data information the module has. For any transformer Π , clearly, the algorithm function $\hat{\pi}$ must be part of the information included in the environment. Secondly, the operator definition for the transformer algorithm $\hat{\pi}$ needs to be collected as well for exporting. Instead of just passing the name of the operator s , we decided to pass the whole definition of the operator to the environment to formalize the name space environment.

This is not going to affect the original understanding of the formal specification of $\Pi = (\pi, \hat{\pi})$ for the rest of system, since the name space environment eventually creates three different kinds of pair relations for exporting, while one of those pair relations reflects exactly the original meaning of Π . Details will be shown in the **Name Space Environment** section of this chapter.

7.2.3 Examples

Next, we present an example that goes through these five steps.

Conjunction

Operator: (`and_e` :: E, E, E)

Definition:

$$\forall e_1, e_2 : E_{fo} . (\text{and_e} :: E, E, E)(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \equiv \neg(\neg\llbracket e_1 \rrbracket \vee \neg\llbracket e_2 \rrbracket).$$

Step 1 : Operator definition

OCaml Code 4 Operator Definition of the `and_e` operator - `logicals.ml`

```
let and_e = build "and_e" 2
```

The function `build` creates the S-expression E_{op} of type **operator** in the form of `(and_e :: E, E, E)` according to the definition of the operator `and_e`.

Step 2 : Constructors for meaning formulas

This constructor builds the expression which is going to be used for expressing the meaning formula for the `and_e` transformer as $\llbracket f_1 \rrbracket_{fo} \wedge \llbracket f_2 \rrbracket_{fo}$.

OCaml Code 5 Constructors for meaning formulas - types.ml

```
let andf (f1:formula) (f2:formula) : formula =
  apply_f Builtin.form_and (map lift [f1;f2])
```

Step 3 : Transformer function

OCaml Code 6 Transformer function for the `and_e` operator - logicals.ml

```
let fn_and_e (el : quoted list) : quoted =
  Impl.two_args el K.And
```

The transformer function `fn_and_e` is quite simple. It just calls the algorithm function `two_args`, which is coded in the implementation module, to perform the algorithm for the transformer `and_e`.

Step 4 : Algorithm implementation

OCaml Code 7 Algorithm implementation for the `and_e` operator - logicals.ml

```
let two_args (el : quoted list) (op : K.keyword) : quoted =
  let (e1,e2) = check_length2 el in
  let res =
    match (C.unquote e1, C.unquote e2, op) with
    .....
    | ((#formula as f1), (#formula as f2), K.And)
    .....
      -> C.form_app_tm (C.k_sym op) [C.kd_form f1; C.kd_form f2]
    .....
  in C.quote (C.lift res)
```

The algorithm implementation of the transformer `and_e` is basically to construct the expression of an `and_e` operator application with two input expressions of type

formula as its arguments, so that the output expression of the `and_e` transformer can be evaluated once it gets to the transformer *simplify*.

Step 5 : Transformer formulation

OCaml Code 8 Transformer formulation for the `and_e` operator - logicals.ml

```
let l1 = [
  .....
  (and_e, fn_and_e);
  ..... ]

let logical = create "logical" LogicalLib.l1 []
```

The list `l1` of the **OCaml Code 8** is used to collect all the pairs of $II = (\pi, \hat{\pi})$ in the module. The pair `(and_e, fn_and_e)` is the one for this example. Then all those pairs will be added into the name space, named as *logical* created particular for this module, during the creation process of the name space environment.

7.3 The Kernel Theory

Section 6 of the formal specification of Chiron [5] lists and defines the fundamental operators of Chiron in three different categories, namely the logical operators, syntac-

tic operators and set-theoretic operators. To form the kernel theory, we implement the first two categories by creating the modules **Logicals** and **Basics**, respectively.

7.3.1 Logicals

The module **Logicals** is the implementation module for the basic logical operators, which are defined in section 6.1 of [5]. In addition, we extend the **Logicals** with some other useful operators.

Table C.1 lists the operators currently implemented in *Chiron*.

7.3.2 Basics

We create the **Basics** module for holding the implementation for the syntactic operators, defined in section 6.3 of [5].

Table D.1 lists the operators currently implemented in *Chiron*.

7.4 Libraries

7.4.1 λ -Calculus

The library, **lambdacalc**, currently contains only one transformer, which is a simple version of beta-reduction. Details will be discussed in Chapter 10.

7.4.2 Natural Numbers

The **Nats** library is an implementation for the theory of the natural numbers at a very basic level.

7.5 Name Space Environment

7.5.1 Name Space Environment Introduction

For any pair of $\Pi = (\pi, \hat{\pi})$, the operator π and the algorithm function $\hat{\pi}$ are implemented independently of each other in one module, without being connected by any kind of data structure. A *Name Space*, implemented by a special data structure *environment*, is created for the purpose of bundling together the various routines contained in an implementation module. We create separate name spaces for the named transformers corresponding to these routines. It also provides all the fundamental utilities for manipulating and maintaining the underlying implementation module. Therefore, the name space for a kernel theory (or a library) serves as the interface for the implementation module by exporting the low level implementations of transformers to clients. Furthermore, it can be extended to a biform theory in a way that not only makes the code clean syntactically at the implementation level, by implementing the idea of a biform theory with several different layers gradually, but also makes the semantic meaning of the notion of a biform theory clear to both the developer and the user.

7.5.2 Name Space Environment Components

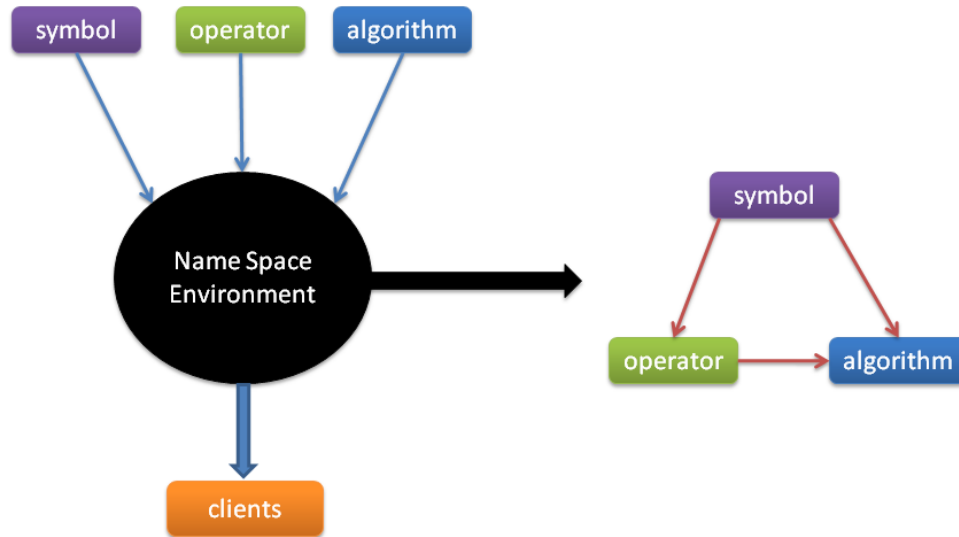


Figure 7.2: Name Space Environment

There are three elementary data elements in a Chiron name space environment. Since a *name space* serves as the interface for exporting the implementation of the transformer Π , two of these data elements are the algorithm $\hat{\pi}$ that implements the (possibly partial) function $f_{\hat{\pi}} : \varepsilon^n \rightarrow \varepsilon$ and the symbol s that serves as a name for the algorithm $\hat{\pi}$. In addition, we include the **operator** type expression $O = (\text{op}, s, k_1, \dots, k_{n+1})$, which indicates both the name and the signature of the transformer, as the third data element for a *name space*.

Instead of putting those three data elements into three separated meaningless lists, we organize them into three binary relations, for the purpose of providing three sorts of search methods the user might want to use.

(1) (**symbol** \rightarrow $\hat{\pi}$)

One of those three kinds of binary relations is the one originally defined in the formal specification of Chiron as $\Pi = (\pi, \hat{\pi})$, namely, a user can search for the algorithm function $\hat{\pi}$ by giving the name of the transformer.

(2) (**symbol** \rightarrow **operator**)

Secondly, the **operator**, which contains not only the symbol s for representing the name of the transformer but also the number of typed arguments required by the transformer, can be retrieved by giving the name of the transformer s , as well.

(3) (**operator** \rightarrow $\hat{\pi}$)

Lastly, the algorithm $\hat{\pi}$ can be looked up by providing the **operator** expression.

These have been coded into three hash tables, as shown in **OCaml Code 9**, defined as types in the system with the name **named_algorithms**, **named_operators** and **op_algorithms**.

OCaml Code 9 Hashtables For Pair Relations - environment.ml

```
type named_algorithms = ( symbol   , algorithm) Hashtbl.t
type named_operators  = ( symbol   , operator ) Hashtbl.t
type op_algorithms    = ( operator, algorithm) Hashtbl.t
```

7.5.3 Name Space Environment Operations

add : Add a new transformer entry to the environment

OCaml Code 10 Add a new transformer entry to the environment - environment.ml

```
let add (e:environment) ('Operator(sy,kl,k) as op) (tr : algorithm) : unit =
  if (List.for_all (fun x -> x = B.expr_kind) (k::kl)) then
    (Hashtbl.add e.operators sy op;
     Hashtbl.add e.algorithms sy tr;
     Hashtbl.add e.trans_from_oper op tr)
  else
    failwith "Transformer Operator should
             take and return Expressions"
```

The `add` function is used for adding new name space entries. It needs three pieces of information, namely the three data elements [**operator**, **algorithm** and **symbol** in Figure 7.2], to complete an insertion operation. Two of them, the **operator** expression and the **algorithm** function, are actually the input arguments of the **Add** function. The last elementary data, which is the **symbol** to represent the name of the operator, can be easily extracted from the operator expression by using the pattern matching feature of OCaml on the second component of an operator expression. Since the data structure of an environment is created with three hash tables to hold the three sorts of pair relations introduced in the previous section, the insertion operation can be done by just calling the built-in `add` routine provided by the hash table module of OCaml to insert these three pairs of data into the three hash tables accordingly.

lookup : Lookup name space elements

There are three sorts of lookup tools available in *Chiron*, one for each pair relation defined at the beginning of this section on page 59.

- Given the name (type **symbol**) of the operator, return the algorithm $\hat{\pi}$ (type

algorithm).

- Given the name (type **symbol**) of the operator, return the operator expression (type **operator**).
- Given the operation expression (type **operator**), return the algorithm $\hat{\pi}$ (type **algorithm**).

merge : Merger of two name spaces

Name Space Mergence can be useful in the following two situations:

- Users may want to use the **merge** tool to either create a larger name space on top of two existing name spaces, or extend an existing name space by importing other name spaces into the current name space. The former case needs a new name for the new larger name space, while in the latter case, one keeps the old name of the current base name space.
- The system itself may want to create a transformer pool of a certain size at some point to automatically merge the related transformers from different name spaces into a single name space for various purposes. For instance,

OCaml Code 11 Mergence of two name spaces - `testbox.ml`

```
let fundamental_env =  
  Environment.merge Basics.basic Logicals.logical "fundamental"
```

we merge the name space environments of **Basics** and **Logicals** to a new name space environment called *fundamental*, which is the name space for the kernel

theory of Chiron.

Chapter 8

Run

This chapter presents one important meta-program developed for our MMS, called *run*, which is used for processing inputs of Chiron expressions with two tasks :

- (1) Check if all sub S-expressions of operator applications occurring in the input expression are known to the system.
- (2) Run the named transformers associated with those known operators.

8.1 Operator Applications

Operator applications in Chiron are defined by the formation rule **P-Expr-2 (Operator application)** from the Chiron paper [5, chap. 3] as below,

P-Expr-2 (Operator application)

$$\frac{\text{operator}[(\text{op}, s, k_1, \dots, k_{n+1})], \text{expr}[e_1], \dots, \text{expr}[e_n]}{\text{p-expr}[(\text{op-app}, (\text{op}, s, k_1, \dots, k_{n+1}), e_1, \dots, e_n) : k_{n+1}]}$$

where $n \geq 0$ and ($k_i = \text{type}$ and $\text{type}[e_i]$), ($\text{type}[k_i]$ and $\text{term}[e_i]$), or ($k_i = \text{formula}$

and $\mathbf{formula}[e_i]$) for all i with $1 \leq i \leq n$.

Based on the type system of *Chiron*, the **Constructors** module provides three different constructors for constructing expressions of operator applications, which are *TypeApplyTerm*, *ApplyTerm* and *FApplyTerm*¹, for different types of operator applications of type **typ**, type **term** and type **formula** respectively. The type of an operator application is decided by the type of the output, namely the kind of k_{n+1} ² in the formation rule **P-Expr-2 (Operator application)**.

However, as the kind for the output of a Chiron transformer, k_{n+1} is always kind **term** since all the input expressions and the output expression of Chiron transformers are of type **quoted**³ strictly, and type **quoted** is a subtype of type **term** as defined in the type system of *Chiron*. Therefore, all operator applications in the S-expressions that are passed to *run* are under the construction of *ApplyTerm*⁴ since all the arguments of those operator applications, $\mathbf{expr}[e_i]$, are really of type **term**⁵, namely $\mathbf{term}[e_i]$ for all i with $1 \leq i \leq n$.

In summary, *run* traverses the tree structure of S-expressions in pre-order to find all possible sub S-expressions of **term** type operator applications. Follow by a *validation* procedure for each operator application and a possible *execution* step if the validation succeeds.

¹See section : Type Declaration - Part 3, on page 31.

²See section : Type Declaration - Part 2, on page 31.

³Page 37.

⁴*ApplyTerm* is the constructor used for building **term** type operator applications.

⁵**quoted** :>**term**.

8.2 Validation

The validation step is responsible for ensuring the existence of the transformer Π in the name space which is passed as one argument of *run*. The validation process comprises calling the `lookup` tool provided by the **Environment** module. The `lookup` tool requires two input arguments, which are the name of the operator of the transformer and the name of the name space. In particular, the name space with the name *env* must be `valid` in our *Chiron* system in order to pass the OCaml type checking system. A name space is `valid` if it is pre-defined and loaded during the *Chiron* initialization process, or it was created previously by the user before the `lookup` call.

`Lookup` searches for the pair $(op, \hat{\pi})$ in the name space *env*. $\hat{\pi}$ is the algorithm function for the operator with the name *op*. Since the pair structure is implemented by using the hash table data structure provided by OCaml from its built-in modules, the `lookup` tool is implemented as a simple OCaml function which performs a standard hash table lookup operation.

If the pair $(op, \hat{\pi})$ exists in the name space, *env*. `Lookup` returns the corresponding algorithm function $\hat{\pi}$, for the operator with the name *op*, i.e., the operator is defined in the name space; and the algorithm function for that operator is coded in the name space which is about to be called. Otherwise, `lookup` returns a special OCaml value, *None*, to inform the *run* that the operator with the name *op* is invalid in the name space *env*.

8.3 Execution

The execution step runs only if the validation process is passed for the subexpression of an operator application. In case of success for the validation process performed by the `lookup` tool provided by the ***Environment*** module, the algorithm $\hat{\pi}$ for the operator application is returned by the `lookup` function call. Since the algorithms are implemented as OCaml functions, `run` executes the sub S-expression e of that operator application by simply calling that algorithm function, $\hat{\pi}$, with all necessary arguments provided by e as parts of its S-expression construction. The subexpression e is then replaced by the expression, e' , which is the return value of the algorithm function $\hat{\pi}$.

Example :

`run e`, where e is an expression of a `beta_reduce` operator application, which can be expressed in lambda calculus form as $(\lambda x. f x) (y)$.

Beta reduction is implemented as a transformer (`beta_reduce`, `fn_beta_reduce`) in the current version of the *Chiron* system in the lambda calculus theory where `beta_reduce` is the name of the operator of the transformer, and `fn_beta_reduce` is the name of the algorithm function of the transformer. (Details in Chapter 10 : Beta Reduction)

In this example, `run` calls the `lookup` tool to search for the beta reduction algorithm function from the name space created for lambda calculus. The algorithm implemented as an OCaml function with the name `fn_beta_reduce` is then returned by the `lookup` procedure. `Run` calls the algorithm function `fn_beta_reduce` with the argument y which is extracted from e to perform the beta reduction.

Finally, the expression $(f\ y)$ is returned by the algorithm function as its output.

$((\lambda x. f\ x)\ y) =_{\text{beta-reduction}} f\ y$ i.e., the original expression e gets replaced by $(f\ y)$.

$\text{run } e \implies e'$, where $e' = f\ y$.

Chapter 9

Simplify

This chapter presents another one of those important transformers developed during the first stage of the development of the MMS, called *simplify* (`simplify :: E, E`), which is used for simplifying S-expressions by replacing complicated formulas by simple logically equivalent formulas. Currently, only boolean algebra simplifications are supported by the system.

9.1 Implementation Module

Simplify traverses the tree structure of input S-expressions in pre-order to find all sub S-expressions which can possibly be simplified by *Chiron*. Then it simplifies those sub S-expressions and replaces them by the simplified S-expressions.

All simplifications are implemented in a submodule of the ***Simplify*** module called ***Impl***. In this approach, we can not only make the implementation efficient, but also provide a better module structure for future development.

(1) Efficiency

By having the implementations for all kinds of simplifications in a separate module, we can implement different algorithms by different routines. Since a complicated simplification algorithm is usually a composition of multiple simple simplifications, we can implement the routine for a complicated simplification algorithm by calling those simple simplification routines as parts of its implementation.

(2) Development

- Simplifications which are of different kinds in terms of semantics may seem to be similar in terms of their implementation. By creating simplification functions in a separate module, we can avoid a lot of possible redundant code for similar simplification implementations. We first create some generic functions, and then, develop the main simplification functions on top of these.
- Instead of implementing the simplifications as parts of the main `Simplify` function, we can give meaningful names for all simplification algorithm routines by implementing them in a separate module, so that the developer can locate the right function as needed easily.

Example :

For a simple simplification of the logical operator `and`.

[pseudocode]

```
Simplify (e) = match e with
```

```

.....
.....
(traversal)
.....
finds the subexpression of an AND application as AND(a, b)
-> option 1 (in-line) : return (a == true) && (b == true)
    option 2 (dispatch) : call the routine
                           written in a separately module
                           to perform the AND simplification.
.....

```

Firstly, the main function of `simplify` finds the subexpressions of `and` applications by pattern matching on the input expression e . Once it finds one, there are two options for a simplification operation. Our system chooses the second option for performing the `and` simplification, by creating the `and` simplification routine separately.

9.2 Boolean Algebra Simplifications

Generally, boolean algebra simplifications are supported in all sorts of MMS by a *simplification tool* similar to the one we created for *Chiron*, “`simplify`”.

The current version of *Chiron* includes an experiment on representing *Church numerals* in *Chiron*. Various simplification tests are tested by using the *Church numerals* module, called `test_input_ch_num`. Details will be discussed and shown in Chapter 12.

Chapter 10

Beta Reduction

Beta reduction is implemented as a Chiron transformer `beta_reduce` in the theory for lambda calculus. In particular, the beta reduction for expressions of type **term** has been carefully implemented and tested.

The operator is created in *Chiron* as : `(beta_reduce :: E, E)`. i.e., the transformer `beta_reduce` takes a Chiron expression as its input, and returns a Chiron expression as its output.

10.1 Implementation

The algorithm function of beta reduction (only for **term** type expressions currently) is implemented in a single OCaml function. The function basically needs to meet the following three requirements :

- Recursively traverse the term searching for *redex*.
- *Prevent* infinite looping.

- *Reduce* the redex.

The implementation achieves the first two goals by using the “pattern matching” feature. It matches the subexpressions of the input expression based on a correct understanding of the syntax of Chiron expressions. Once the redex is found, the third goal can be accomplished by applying the beta reduction rule to the redex.

10.1.1 Expression Syntax

Any expression e is one of the following :

- x - a variable.
- $\lambda x.e$ - a lambda abstraction.
- $e_1 e_2$ - an application.

If $E_1 \rightarrow_\beta E_2$, then the reduction must be one of those cases :

$$\frac{\frac{(e e_1) \rightarrow_\beta (e e_2)}{(e_1 e) \rightarrow_\beta (e_2 e)}}{(\lambda x.e_1) \rightarrow_\beta (\lambda x.e_2)}}$$

Table 10.1: Patterns of $E_1 \rightarrow_\beta E_2$ may involve a beta reduction

Therefore, we only pattern match the expression on the above three patterns for a possible reduction action; for all the other patterns, the function just returns the original expression immediately.

10.1.2 Redex

In the lambda calculus, a beta redex is a term of the form:

$$(\lambda x.b)(a)$$

where the term b may or may not involve the variable x .

An expression of the form $((\lambda x.e_1) e_2)$ is called a redex (reducible expression). Thus, in addition to the three cases listed in table 10.1, there is one more case needs to be pattern matched for applying the beta reduction rule.

10.1.3 Infinite Looping

For the first two cases in Table 10.1, if both the inside subexpressions cannot be reduced further, the expression as a whole should stop the recursive pattern matching process to prevent *infinite looping*.

OCaml Code 12 Prevent Infinite Looping

[pseudo code]

```

reduce_term e = match e with (* Pattern Matching on the expression *)
  ...
  | (e1, e2) -> (* Pattern Matching for the first 2 cases *)
    (* Try to reduce the expression inside *)
    let e1_reduced = reduce_term e1 in
    let e2_reduced = reduce_term e2 in

    if (e1 = e1_reduced) AND (e2 = e2_reduced) then
      return e (* Can not be reduced further, stop the recursive looping *)
    else
      reduce_term (e1_reduced, e2_reduced)

```

For instance, the if condition clause in the **OCaml Code** 12 stops the recursive looping for the recursive reduction function `reduce_term` when both the inside subexpressions `e1` and `e2` can not be reduced further.

10.1.4 Reduce the Redex

To reduce a beta redex in the form of $(\lambda x.b)(a)$, we call the `sub` transformer defined in the kernel theory to do the substitution for the variable a in the expression b . The substitution is defined in table 10.2¹ to replace free occurrences (the set of free variables in b , denoted as $FV(b)$, is defined in table 10.3) of x in b with a (written as $[a/x]b$).

• $[a/x]x = a$
• $[a/x]y = y$
• $[a/x](e_1 e_2) = ([a/x]e_1) ([a/x]e_2)$
• $[a/x](\lambda x.b) = \lambda x.b$
• $[a/x](\lambda y.b) = \lambda y.b$, if $x \notin FV(b)$
• $[a/x](\lambda y.b) = \lambda y.[a/x]b$, if $x \in FV(b)$, $y \notin FV(a)$
• $[a/x](\lambda y.b) = \lambda z.[a/x][z/y]b$, if $x \in FV(b)$, $y \in FV(a)$

Table 10.2: Substitution

$FV(b) = b$
$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$
$FV(\lambda x.e) = FV(e) - x$

Table 10.3: Free Variable

¹*Alpha Renaming* has not been ingrated with the current version of Chiron beta reduction.

10.2 Tests

As we have introduced in the **Simplify** chapter, our MMS includes an experiment on representing *Church numerals* in *Chiron*. *Church numerals* are well suited for testing the current implementation of beta reduction for Chiron. Details will be discussed and shown in Chapter 12.

Chapter 11

Biform Theory

The notion of a biform theory is one of those most important creative points in FFMM [6] by providing a formal context in which deduction and computation can be merged.

Chiron is an exceptionally well-suited logic for formalizing biform theories since it has a high level of both theoretical and practical expressivity. Precisely, the meaning formulas of rules can be directly expressed in Chiron.

For my part of *Chiron* development as my graduate work, the definition of the notion of a biform theory has been coded on a very basic basis with three experiments for the three name space environments shown in the second half of Chapter 7.

11.1 Definition

11.1.1 Biform Theories in Chiron

A *biform theory* in a general logic \mathbf{K} is a triple $T = (L, \Gamma, \Omega)$, where

- L is the language of \mathbf{K} . (\mathbf{K} is Chiron for our MMS) i.e., a set of **operators**. There are two sorts of operators in T .
 - (1) *transformer-less* operators which are representing the **concepts** of T . They do not require to have transformers associated with.
 - (2) *transformer* operators which specify the name of the **transformer** in type **symbol** and the type of expressions for both the inputs and the output of the transformers.
- The members of Γ are the **axioms** of T . The axioms specify the meanings of the concepts and the transformers of T . Since an axiom specifies the semantic relationship between the inputs and the output of the algorithm of the transformer, we also name the axiom as **meaning formula**, M , in Chiron.
- Ω is a set of **transformers** for L . The definition of a transformer Π which is a pair of $(\pi, \hat{\pi})$ in T has been explained at the beginning of Chapter 7 on page 48, where π is a **symbol** used to represent the name of the transformer, and $\hat{\pi}$ is the corresponding algorithm of the transformer.

11.1.2 Rules

A **rule** in L is a pair $R = (\Pi, M)$ where

- Π is the n -ary transformer in L .
- M is a formula that uses π to relate the values of the inputs to $\hat{\pi}$ to the value of the output of $\hat{\pi}$.

Proposition 11.1.1 For every transformer operator with the name π in T , Chiron has exactly one corresponding transformer algorithm $\hat{\pi}$ and has n meaning formula(s), where $n \geq 1$, to specify the semantic meaning of the transformer.

11.2 Chiron Representation

For every theory in Chiron, the *transformers* of the theory have been implemented in a separated implementation module firstly; and the implementation of those *transformers* then gets exported in a name space which is created on top of the implementation module. The current version of *Chiron* contains four *name spaces*. Two name spaces, **Basics** and **Logicals**, are created for composing the kernel theory. Two name spaces for the theory of the natural numbers and lambda calculus, **Nats** and **lambdacalc** respectively.

In Chapter 7, **Name Space**, we have shown that a name space environment is a well-designed interface for bundling together the low level implementations for the named transformers for each biform theory. The transformers of each theory have been organized into pair relations by using the hash table data structure, and then, a biform theory from an existing *Chiron* name space can just be constructed by adding

- (1) *transformer-less operators* for representing the base elements of the theory, such as constants, types.
- (2) **Formulas** in L viewed as transformer-less rules.
- (3) **Formulas** in L viewed as the meaning formulas for the transformers of the theory.

In the next section, I create a simple biform theory of (high-order) Peano arithmetic to show the formalization of a nontrivial biform theory in *Chiron* step by step. The example was previously sketched in words in the paper of *Biform Theories in Chiron*. [3, pg. 12]

11.3 Biform Theory of Peano Arithmetic

The following abbreviations have been used in this example while both the module **Types** and **Environments** are opened for directly inheritance.

OCaml Code 13 Module Abbreviations

```
module K = Keywords
module C = Types.Constructors.Raw
module M = Types.Constructors.Main
module B = Types.Builtin
module P = Pa.PaLib
module Bi = Biform
module U = Biform_utilities
```

We would firstly sketch the development of the *biform theory* for Peano Arithmetic briefly, and then, continue with the example fully.

- Create the type **nat** of natural numbers (as 0-ary operator).
- Create $(0 : \mathbf{nat})$ (as 0-ary operator) to represent the constant 0.
- Create the transformer operator for the **successor** function by coding the transformer Π_{suc} , and then adding the meaning formulas for expressing the axioms related to it.

- Add the meaning formulas for expressing the induction axiom of Peano Arithmetic.
- Create the transformer operators 1 , $+$, \times for 1 , the addition function $+$, and the multiplication function respectively with the same approach as we do for the `successor` function.

The development of the *biform theory* T of Peano arithmetic in *Chiron* starts with the base operator `nat` which specifies the type of the natural number.

- (1) L contains operator `nat_e` that represents the type of natural number.

OCaml Code 14 operator `nat` - `pa_bif.ml`

```
let nat_e  : operator = M.create_type "nat"
let nat_typ : typ    = B.type_from_oper nat_e
```

- (2) L contains operator `0` that represents the constant 0 , which is assumed to be a natural number in PA.

OCaml Code 15 operator `zero` - `pa_bif.ml`

```
let zero_e : operator = C.signature sym_zero [] (B.kind_from_type nat_typ)
let zero   : term     = M.apply sym_zero P.nat_typ
```

For both the operator `nat_e` and the operator `0`, there is no transformer created

for any of them since they are *transformer-less* operators which represent the concepts of T .

- (3) On the other hand, Γ contains a series of axioms (meaning formulas) for defining the properties of natural numbers.

$\forall x, x \in \mathbf{nat}. x = x$ i.e., equality is reflexive.
$\forall x, y \in \mathbf{nat}. x = y \supset y = x$ i.e., equality is symmetric.
$\forall x, y, z \in \mathbf{nat}. (x = y) \wedge (y = z) \supset x = z$ i.e., equality is transitive.
$\forall x, y. x \in \mathbf{nat} \wedge (x = y) \supset y \in \mathbf{nat}$ i.e., the natural numbers are closed under equality.

Table 11.1: A Series of Axioms for Defining The Properties of Natural Numbers

- (4) L contains another important operator S which represents the successor function. By having the constant 0 as the *first* natural number, the natural numbers are assumed to be closed under the successor function. The successor operator is the first transformer operator in our Peano Arithmetic example which has the algorithm of taking one arbitrary natural number n as its input and returns the natural number $n + 1$.

OCaml Code 16 `suc_e` operator - `pa.ml`

```
let suc_e = build "suc_e" 1
```

The function `build` creates the S-expression E_{op} of type **operator** in the form of $(\mathbf{sub_e} :: E_{nat}, E_{nat})$.

- (5) By having the successor function, we are able to write more meaning formulas to express more axioms of natural numbers :

$\forall x, x \in \text{nat. } \text{suc}(x) \in \text{nat}$
$\forall x, x \in \text{nat. } \text{suc}(x) \neq 0$
$\forall x, y \in \text{nat. } \text{suc}(x) = \text{suc}(y) \supset x = y$ i.e., successor function is <i>injective</i> .
If K is a set . $0 \in K \wedge \forall x, x \in \text{nat. } x \in K \supset \text{suc}(x) \in K$, Then K contains every natural numbers.

Table 11.2: More Axioms for The Natural Numbers

- (6) We introduce the operator 1 to represent the natural number $1 = S(0)$ as one example of representing natural numbers by using the successor function. The **OCaml Code 17** is the algorithm function used for constructing the Chiron expression $S(0)$, namely the natural number 1.

OCaml Code 17 natural number 1 - pa.ml

```
let one () : term =
  let zero_tm = M.apply sym_zero nat_typ in
  M.apply_pa sym_suc_e zero_tm nat_typ
```

- (7) Lastly, we would like to add two basic arithmetic operators, + and \times , to represent the addition function and the multiplication function respectively.

i.e., the addition function is defined recursively as

$$a + 0 = a$$

$$a + S(b) = S(a + b)$$

and the multiplication is defined recursively as

$$a \times 0 = 0$$

$$a \times S(b) = a + (a \times b)$$

The meaning formulas of the two arithmetics are expressed as

$$\forall e_1, e_2 : E_{te} . e_1 \downarrow \text{nat} \wedge e_2 \downarrow \text{nat} \supset$$

$$\llbracket (\text{plus_e} :: E_{te}, E_{te}, E_{te})(e_1, e_2) \rrbracket_{te} \equiv (\text{fn_plus_e} :: \text{term}, \text{term}, \text{term})(\llbracket e_1 \rrbracket_{te}, \llbracket e_2 \rrbracket_{te})$$

$$\forall e_1, e_2 : E_{te} . e_1 \downarrow \text{nat} \wedge e_2 \downarrow \text{nat} \supset$$

$$\llbracket (\text{mult_e} :: E_{te}, E_{te}, E_{te})(e_1, e_2) \rrbracket_{te} \equiv (\text{fn_mult_e} :: \text{term}, \text{term}, \text{term})(\llbracket e_1 \rrbracket_{te}, \llbracket e_2 \rrbracket_{te})$$

Chapter 12

Testing : Church Numerals in Chiron

Church numerals are representations of natural numbers using lambda notation under Church encoding [10, Chap 2], which is a means of embedding data and operators into the lambda calculus. Integers, booleans, and boolean arithmetic are mapped to higher-order functions under Church encoding. Because of the heavy use of lambda notation, the implementation of Church numerals is a very good choice to test programs involve lambda constructions.

This chapter presents our work in two parts¹ :

- (1) Representing Church numerals in Chiron data structure.
- (2) Testing the implementation of beta reduction in Chiron.

¹All related code discussed for these two parts are located in `test_input_ch_num.ml`

12.1 Chiron Representation

Natural numbers

We first describe Church's representation of the natural numbers in the lambda calculus. Zero is represented as

$$0 \equiv \lambda s. \lambda z. z.$$

Then the natural number n ($n > 0$) is represented by the higher-order function which maps any other function f to its n -fold composition :

$$1 \equiv \lambda s. \lambda z. s z$$

$$2 \equiv \lambda s. \lambda z. s (s z)$$

$$3 \equiv \lambda s. \lambda z. s (s (s z))$$

...

$$n \equiv \lambda s. \lambda z. s^n z$$

Hence, natural numbers are represented as a Chiron data structure as follows :

natural number	lambda term	Chiron representation
		$\mathbf{s} \equiv (\text{var}, s, \text{class}), \mathbf{z} \equiv (\text{var}, z, \text{class})$
0	$\lambda s. \lambda z. z$	$(\text{fun-abs}, \mathbf{s}, (\text{fun-abs}, \mathbf{z}, \mathbf{z}))$
1	$\lambda s. \lambda z. s z$	$(\text{fun-abs}, \mathbf{s}, (\text{fun-abs}, \mathbf{z}, (\text{fun-app}, \mathbf{s}, \mathbf{z})))$
2	$\lambda s. \lambda z. s (s z)$	$(\text{fun-abs}, \mathbf{s}, (\text{fun-abs}, \mathbf{z}, (\text{fun-app}, \mathbf{s}, (\text{fun-app}, \mathbf{s}, \mathbf{z}))))$
...

Table 12.1: Natural Numbers

By looking at the first three examples of the Chiron representation of natural numbers, we notice that the Chiron representation of $(n + 1)$ is constructed from the Chiron

representation of n simply by replacing the last occurrence of the \mathbf{z} by $(\text{fun-app}, \mathbf{s}, \mathbf{z})$.

Successor

One of the fundamental functions in Church numerals is the successor function, $\text{succ} \equiv \lambda n. \lambda s. \lambda z. s (n s z)$.

Number Arithmetic

Number arithmetic can be represented in *Chiron* similarly by directly translation of lambda terms.

Abbreviations: $\mathbf{s} \equiv (\text{var}, s, \text{class})$, $\mathbf{z} \equiv (\text{var}, z, \text{class})$, $\mathbf{m} \equiv (\text{var}, m, \text{class})$, $\mathbf{n} \equiv (\text{var}, n, \text{class})$

Addition

$$\text{Addition} \equiv \lambda m. \lambda n. \lambda s. \lambda z. (((m \text{ succ}) n) s) z$$

Multiplication

$$\text{Multiplication} \equiv \lambda m. \lambda n. \lambda s. (m (n s))$$

Power

$$\text{Power} \equiv \lambda m. \lambda n. (m n)$$

Church Booleans

Church booleans are formally defined in the lambda calculus as

$$\text{true} \equiv \lambda t. \lambda f. t$$

$$\text{false} \equiv \lambda t. \lambda f. f$$

Boolean Arithmetic

Common boolean functions are implemented in *Chiron* as

AND

$$\text{and} \equiv \lambda b. \lambda c. ((b\ c)\ \text{false})$$

OR

$$\text{or} \equiv \lambda b. \lambda c. ((b\ \text{true})\ c)$$

NOT

$$\text{not} \equiv \lambda b. ((b\ \text{false})\ \text{true})$$

Table 12.2 introduces the compact notation for the names of variables and functions defined in *Chiron* for Church Numerals which we will use for the rest of paper whenever it is convenient.

Compact Notation	Names in Code	Representing
0_{ch}	ch_zero	natural number 0
n_{ch}	ch_n	natural number n
succ	ch_next	successor function, (succ n) \rightarrow n+1
ch_gen	ch_fx	generic function for constructing the natural number, n. (ch_gen n) \rightarrow n_{ch}
T_{ch}	ch_tru	$\lambda x.\lambda y. x$
F_{ch}	ch_fls	$\lambda x.\lambda y. y$
\wedge_{ch}	chur_and	\wedge
\vee_{ch}	chur_or	\vee
NOT_{ch}	chur_not	\neg
IF_{ch}	chur_if	if - boolean conditional operator
$+_{ch}$	ch_add	addition - number arithmetic operator
$*_{ch}$	ch_mult	multiplication - number arithmetic operator
exp_{ch}	ch_power	exponentiation - number arithmetic operator

Table 12.2: Compact Notation for Church Numerals in *Chiron*

12.2 Tests for Beta Reduction

Since Church encoding is a means of embedding data and operators into the lambda calculus, the Church numerals, as one of those most familiar examples based on Church encoding, form an exceptionally well-suited data structure for testing the beta reduction of *Chiron*.

Example 1 :

$$\begin{aligned}
\text{succ } 1 &\equiv_{\text{definition expansion}} (\lambda n. \lambda f. \lambda x. (f ((n f) x)) 1) \\
&\equiv_{\eta\text{-reduction}} (\lambda n. \lambda f. \lambda x. (f ((n f) x)) (\lambda f. \lambda x. f x)) \\
&\equiv_{\beta\text{-reduction}} (\lambda f. \lambda x. (f (((\lambda f. \lambda x. f x) f) x))) \\
&\equiv_{\beta\text{-reduction}} (\lambda f. \lambda x. (f ((\lambda x. f x) x))) \\
&\equiv_{\beta\text{-reduction}} (\lambda f. \lambda x. (f (f x))) \\
&\equiv 2
\end{aligned}$$

Example 2 :

Note: use $(\lambda s. \lambda x. (s (s (s x))))$ to represent 3 instead of $(\lambda s. \lambda z. (s (s (s z))))$ to avoid name capture, since alpha convention has not been integrated with the beta reduction of *Chiron* yet.

$$\begin{aligned}
mult_{ch} \ 2 \ 3 &\equiv_{definition \ expansion} ((\lambda m. \lambda n. \lambda z. (\mathbf{m} (\mathbf{n} \mathbf{z})) 2) 3) \\
&\equiv_{\beta\text{-reduction}} (\lambda n. \lambda z. (2 (\mathbf{n} \mathbf{z})) 3) \\
&\equiv_{\beta\text{-reduction}} \lambda z. (2 (3 \mathbf{z})) \\
&\equiv_{definition \ expansion} \lambda z. (2 ((\lambda s. \lambda x. (s (s (s x)))) \mathbf{z})) \\
&\equiv_{\beta\text{-reduction}} \lambda z. (2 (\lambda x. (z (z (z x))))) \\
&\equiv_{definition \ expansion} ((\lambda s. \lambda x. (s (s x))) (\lambda x. (z (z (z x))))) \\
&\equiv_{\beta\text{-reduction}} \lambda z. (\lambda x. (\lambda x. (z (z (z x)))) ((\lambda x. (z (z (z x)))) x)) \\
&\equiv_{\beta\text{-reduction}} \lambda z. (\lambda x. (\lambda x. (z (z (z x)))) ((z (z (z x))))) \\
&\equiv_{\beta\text{-reduction}} \lambda z. (\lambda x. (z (z (z ((z (z (z x)))))))) \\
&\equiv 6
\end{aligned}$$

All the test cases listed in Table 12.3 have been successfully tested in *Chiron*.

Testing : Church Numerals in Chiron			
Natural Numbers			
	succ	0	\equiv 1
succ	(succ	(succ 0))	\equiv 3
	ch_{gen}	0	\equiv 0_{ch}
			\dots
	ch_{gen}	6	\equiv 6_{ch}
Church Booleans			
	T_{ch}	x y	\equiv x
	F_{ch}	x y	\equiv y
T_{ch}	T_{ch}	T_{ch}	\equiv T_{ch}
T_{ch}	F_{ch}	F_{ch}	\equiv F_{ch}
Boolean Arithmetics			
T_{ch}	\wedge_{ch}	T_{ch}	\equiv T_{ch}
T_{ch}	\wedge_{ch}	F_{ch}	\equiv F_{ch}
F_{ch}	\wedge_{ch}	F_{ch}	\equiv F_{ch}
T_{ch}	\vee_{ch}	T_{ch}	\equiv T_{ch}
T_{ch}	\vee_{ch}	F_{ch}	\equiv T_{ch}
F_{ch}	\vee_{ch}	F_{ch}	\equiv F_{ch}
	\neg_{ch}	T_{ch}	\equiv F_{ch}
	\neg_{ch}	F_{ch}	\equiv T_{ch}
Number Arithmetics			
$+_{ch}$	1_{ch}	2_{ch}	\equiv 3_{ch}
$+_{ch}$	2_{ch}	2_{ch}	\equiv 4_{ch}
$+_{ch}$	0_{ch}	0_{ch}	\equiv 0_{ch}
$*_{ch}$	0_{ch}	0_{ch}	\equiv 0_{ch}
$*_{ch}$	1_{ch}	3_{ch}	\equiv 3_{ch}
$*_{ch}$	2_{ch}	3_{ch}	\equiv 6_{ch}
$*_{ch}$	0_{ch}	6_{ch}	\equiv 0_{ch}
exp_{ch}	2_{ch}	1_{ch}	\equiv 2_{ch}
exp_{ch}	2_{ch}	2_{ch}	\equiv 4_{ch}
exp_{ch}	2_{ch}	3_{ch}	\equiv 8_{ch}
exp_{ch}	4_{ch}	3_{ch}	\equiv 64_{ch}

Table 12.3: Testing : Church Representation in Chiron

Chapter 13

User-defined Transformers

13.1 Compiler

This small *compiler* is just used for creating new user defined transformers. Users can define their own transformer by calling the main function of the **Compiler** module, called `create_tr`.

```
create_tr (name1 : string, algorithm2 : term) : unit
```

Since all *transformers* need to be retrieved later from a *name space*, another function, called `add_to_env`, is used to accomplish the work of adding the new user defined *transformer* to a *name space*.

¹The name of the transformer is optional, however, it should be the first argument of the `create_tr` function if it is provided by users. If the user does not name the transformer, the `create_tr` function generates a random name automatically for the new transformer by calling the name generator function.

²The algorithm function, $\hat{\pi}_{user_define}$, is required for creating a new user defined transformer. It should be provided by the user, written in lambda form.

```
add_to_env (name3 : string) (env4 : environment) : unit
```

Example : Creating the new transformer with the operator, called “union”, for the *Basics* name space.

Note : Code below is running at low implementation level of *Chiron* in OCaml syntax, an end user should be able to enter all those data and the commands in a high level well-designed GUI in the future. The term, `user_union_input` is math equivalent to $\lambda x.C.(\lambda y.C.(union :: C, C, C) (x, y))$

```
let user_union_input : term =
  let tm = C.apply (C.s_sym "union")
    [C.kd_term (C.variable (C.s_sym "x") class_typ) class_typ;
     C.kd_term (C.variable (C.s_sym "y") class_typ) class_typ]
    class_typ in
  let abs_y = C.fabs (C.s_sym "y") class_typ tm in
  C.fabs (C.s_sym "x") class_typ abs_y

create_tr name:"union" user_union_input

add_to_env Basics.basic "union"
```

³The name of the transformer.

⁴The name of the name space. Importantly, the name space argument provided for the `add_to_env` function needs to be valid before the `add_to_env` call. i.e., *Chiron* only allows users to add new *transformer* to an existing *name space* environment.

The **term** type variable, `user_union_input`, represents the algorithm for the `union` operator which is defined as

$$(\lambda x \lambda y. \text{union}(x, y))$$

in lambda calculus form. The creation of the new `union` transformer, Π_{union} , is done by calling the `create_tr` function by providing the name of the operator, `union`, and the algorithm of the new union transformer, `user_union_input`⁵. In order to use it, we call the `add_to_env` function to add Π_{union} into the kernel theory, **Basics**.

⁵The creation will be failed if the **algorithm** does not exist in the system, and a warning message will be returned by the creation function to the user.

Chapter 14

Conclusion

In this thesis, we have shown the first stage of the development of a *mechanized mathematics system* (MMS) based on a formal framework that integrates and generalizes symbolic computation and formal deduction. The central idea of the framework, developed as part of the MathScheme project at McMaster University, consists the notion of a biform theory, which is simultaneously an axiomatic theory and an algorithmic theory, provides a formal context for both deduction and computation. In order to utilize biform theories to represent mathematics, Chiron is the logic we used for our MMS so that biform theories can be expressed directly.

The development starts with the design of the base type system for the MMS. We designed and implemented the typed type system for *Chiron* by using the strong static typing programming language, OCaml. The current type system categories the type of all expressions into two major types, proper expressions and improper expressions. There are four different kinds of proper expressions, which are **operators**, **types**, **terms** and **formulas**, defined in *Chiron* based on the total mapping between the values in NBG set theory and the expressions in Chiron. On the other hand, four kinds of expressions are defined for

improper expressions.

Secondly, since the type system of *Chiron* is hidden from the end user for the purpose of preventing all sorts of unexcepted system harmfulness, we created all necessary constructor functions in the ***Constructors*** module for translating user input expressions to expressions constructed in Chiron data structure. The module acts as a black box for bridging the expressions that are input by the user and the expressions recognized by our MMS. Importantly, we ensure that the output expression in Chiron data structure, constructed through the ***Constructors*** module, is unique for any unique input expression.

Then, we start the formalization of biform theories of Chiron in two steps.

- Firstly, for all of the transformers in a theory, a name space is created for each theory as an interface for exporting the low level implementations of both the operators and the algorithm functions of the transformers. The name space environment organizes the operator and the algorithm of a transformer in pair relation and provides three different lookup tools for various purposes to the user for quick access. Name spaces can be merged into larger name spaces upon user's needs. The name spaces for kernel theories of Chiron have been completed while the name spaces for both the theory of the natural numbers and the lambda calculus are implemented at a very basic level.
- Secondly, in order to form the biform theories in *Chiron* fully, we add the implementation for those non-transformable operators to represent the concepts of the theory; and finally, express the meaning formulas for all the concepts and the transformers of the theories. The four theories mentioned in the first step have all been extended to biform theories.

Transformers in biform theories need to be recognized and then able to be executed by the MMS, the meta-tool called `run` is created for this purpose. It traverses the input expression to find all possible subexpressions of transformer operator applications. For

each operator application, `run` first does a validation procedure used to make sure that the transformer has been properly defined in the system; and then replaces the subexpression of the operator application by the output expression returned from the algorithm function of the transformer.

There are two important transformers have been implemented separately.

- A simple version of `simplify` is included in our MMS for boolean algebra simplifications.
- Beta reduction is implemented as an example of creating non-built-in transformers on top of the Chiron kernel theories. Beta reduction of Chiron has been tested with the *Chiron* representation of Church numerals successfully.

Finally, all the work documented in this thesis have been fully tested by our testing modules. Since the amount of the testing work is expected to be large enough, we create different modules to prepare tests, create tests, run the tests and report statistic separately.

Appendix A

Compact Notation

Compact notation for proper expressions from [5]. The first group of definitions in table A.1 defines the compact notation for each of the 13 proper expression categories.

Compact Notation	Official Notation
$(s :: k_1, \dots, k_{n+1})$	$(\text{op}, s, k_1, \dots, k_{n+1})$
$(s :: k_1, \dots, k_{n+1})(e_1, \dots, e_n)$	$(\text{op-app}, (\text{op}, s, k_1, \dots, k_{n+1}), e_1, \dots, e_n)$
$(x : \alpha)$	(var, x, α)
$\alpha(a)$	$(\text{type-app}, \alpha, a)$
$(\Lambda x : \alpha . \beta)$	$(\text{dep-fun-type}, (\text{var}, x, \alpha), \beta)$
$f(a)$	$(\text{fun-app}, f, a)$
$(\lambda x : \alpha . b)$	$(\text{fun-abs}, (\text{var}, x, \alpha), b)$
$\text{if}(A, b, c)$	(if, A, b, c)
$(\exists x : \alpha . B)$	$(\text{exist}, (\text{var}, x, \alpha), B)$
$(\iota x : \alpha . B)$	$(\text{def-des}, (\text{var}, x, \alpha), B)$
$(\epsilon x : \alpha . B)$	$(\text{indef-des}, (\text{var}, x, \alpha), B)$
$\llbracket e \rrbracket$	(quote, e)
$\llbracket a \rrbracket_k$	(eval, a, k)
$\llbracket a \rrbracket_{\text{ty}}$	$(\text{eval}, a, \text{type})$
$\llbracket a \rrbracket_{\text{te}}$	$(\text{eval}, a, (\text{op-app}, (\text{op}, \text{class}, \text{type})))$
$\llbracket a \rrbracket_{\text{fo}}$	$(\text{eval}, a, \text{formula})$

Table A.1: Compact Notation

The next group of definitions in table A.2 defines additional compact notation for the built-in operators and the universal quantifier [5].

Compact Notation	Defining Expression
V	$(\text{set} :: \text{type})()$
C	$(\text{class} :: \text{type})()$
E	$(\text{expr} :: \text{type})()$
E_{sy}	$(\text{expr-sym} :: \text{type})()$
E_{op}	$(\text{expr-op} :: \text{type})()$
E_{ty}	$(\text{expr-type} :: \text{type})()$
E_{te}	$(\text{expr-term} :: \text{type})()$
E_a	$(\text{expr-term-type} :: E_{\text{ty}}, \text{type})(a)$
E_{fo}	$(\text{expr-formula} :: \text{type})()$
$(a \in b)$	$(\text{in} :: V, C, \text{formula})(a, b)$
$(\alpha =_{\text{ty}} \beta)$	$(\text{type-equal} :: \text{type}, \text{type}, \text{formula})(\alpha, \beta)$
$(a =_{\alpha} b)$	$(\text{term-equal} :: C, C, \text{type}, \text{formula})(a, b, \alpha)$
$(a = b)$	$(a =_C b)$
$(A \equiv B)$	$(\text{formula-equal} :: \text{formula}, \text{formula}, \text{formula})(A, B)$
$(\neg A)$	$(\text{not} :: \text{formula}, \text{formula})(A)$
$(a \notin b)$	$(\neg(a \in b))$
$(a \neq b)$	$(\neg(a = b))$
$(A \vee B)$	$(\text{or} :: \text{formula}, \text{formula}, \text{formula})(A, B)$
$(\forall x : \alpha . A)$	$(\neg(\exists x : \alpha . (\neg A)))$

Table A.2: Additional Compact Notation

Appendix B

Chiron Types

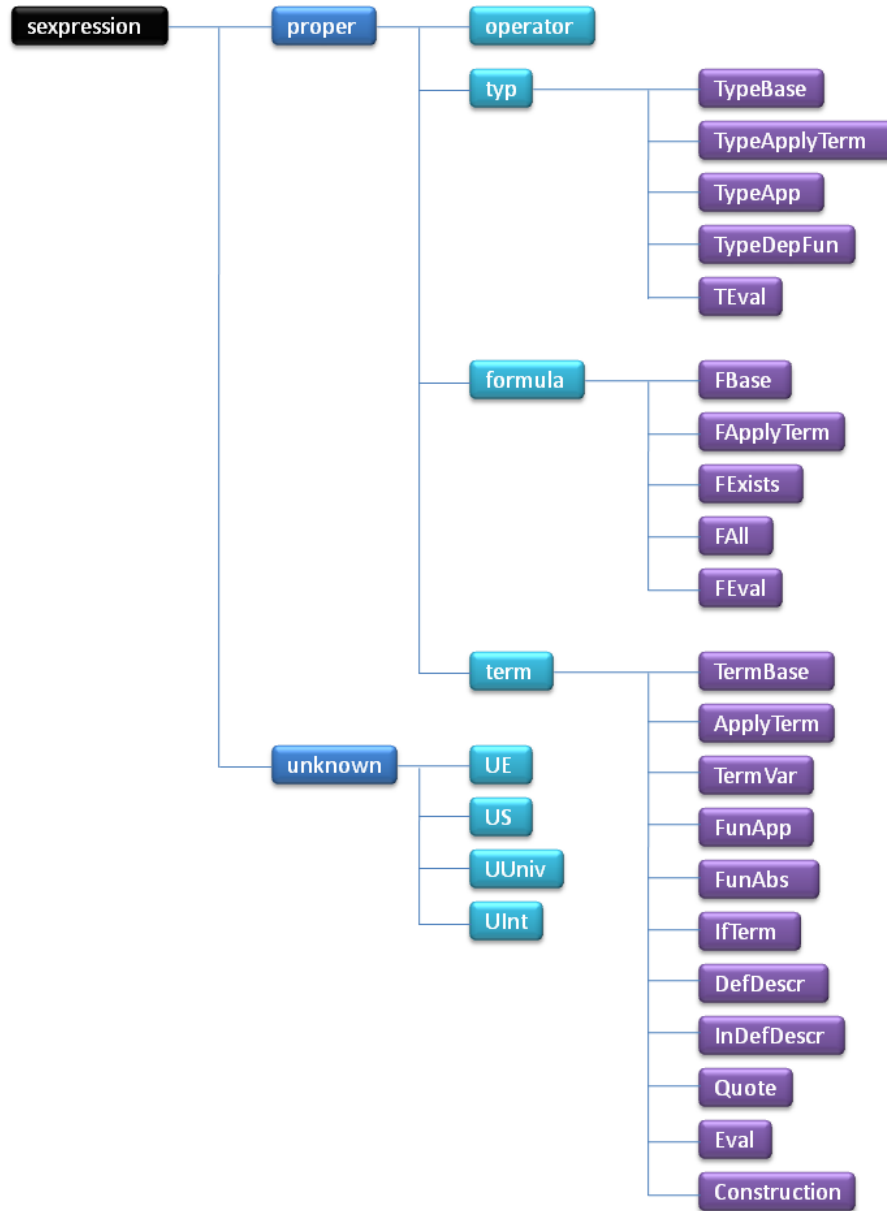


Figure B.1: Chiron Types

Appendix C

Logicals Library

Table C.1: *Logicals* Library

Operator $(\pi, \hat{\pi})$	[main]Impl. Func. for the algorithm $\hat{\pi}$
(and :: formula, formula, formula) (and_e, fn_and_e)	two_args
(or :: formula, formula, formula) (or_e, fn_or_e)	two_args
(not :: formula, formula) (not_e, fn_not_e)	one_arg
(implies :: formula, formula, formula) (implies_e, fn_implies_e)	two_args
(type-equal :: type, type, formula) (type_equal_e, fn_type_equal_e)	two_args
(term-equal :: term, term, type, formula)	

(term_equal_e, fn_term_equal_e)	three_args
(equal :: term, term, type, formula) (equal_e, fn_equal_e)	three_args
(not-equal :: term, term, type, formula) (not_equal_e, fn_not_equal_e)	three_args
(formula-equal :: formula, formula, formula) (formula_equal_e, fn_formula_equal_e)	two_args
(in :: term, term, formula) (is_in_e, fn_is_in_e)	two_args
(not-in :: term, term, formula) (not_in_e, fn_not_in_e)	two_args
(defined-in :: term, type, formula) (defined_in_e, fn_defined_in_e)	two_args

Appendix D

Basics Library

Table D.1: *Basics* Library

Operator $(\pi, \hat{\pi})$	[main]Impl. Func. for the algorithm $\hat{\pi}$
(is-p-expr :: E, formula) $(\text{is_p_expr}, \text{fn_is_p_expr})$	is_proper
(1st-comp :: E, E) $(\text{first_comp}, \text{fn_first_comp})$	get_comp_expr
(is-op :: E, formula) $(\text{is_op}, \text{fn_is_op})$	is_op
(is-var :: E, formula) ^[1] $(\text{is_var}, \text{fn_is_var})$	is_var
(is-type-eqn :: E, formula) $(\text{is_type_eqn}, \text{fn_is_type_eqn})$	is_type_eqn
(is-empty-set :: E, formula)	

(is_empty_set, fn_is_empty_set)	is_empty_set
(is-binder :: E, formula) (is_binder, fn_is_binder)	is_binder
(binder-var :: E, E) ^[2] (binder_var, fn_binder_var)	-
(is-fun-redex :: E, formula) (is_fun_redex, fn_is_fun_redex)	is_fun_redex
(is-fun-type-redex :: E, formula) (is_fun_type_redex, fn_is_fun_type_redex)	is_fun_type_redex
(is-redex :: E, formula) (is_redex, fn_is_redex)	is_redex
(var-sim :: E, E, formula) (var_sim, fn_var_sim)	-
(is-eval-free :: E, formula) (is_eval_free, fn_is_eval_free)	is_eval_free
(free-in :: E, E, formula) (free_in, fn_free_in)	free_in_expression
(free-for :: E, E, E, formula) (free_for, fn_free_for)	free_for_expression
(sub :: E, E, E, formula) (sub, fn_sub)	sub_expression

[1]: Checkers for the other 12 proper expression categories are defined and implemented in a similar way: is-op-app, is-var, is-type-app, is-dep-fun-type, is-fun-app, is-fun-abs, is-if, is-exist, is-def-des, is-indef-des, is-quote, is-eval.

[2]: Selectors for a binder name and a binder body are defined in a similar way:
binder-name and binder-body.

Bibliography

- [1] Jacques Carette and William M. Farmer. High-level theories. *CalcuIemus*, 2008.
- [2] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. Maple v language reference manual. *Spring-Verlag*, 1991.
- [3] W. M. Farmer. Biform theories in Chiron. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer-Verlag, 2007.
- [4] W. 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.
- [5] W. M. Farmer. Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report No. 38, McMaster University, 2007. Revised 2008.
- [6] W. M. Farmer and M. von Mohrenschildt. An overview of a formal framework for managing mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.

-
- [7] William M. Farmer. Stmm: A set theory for mechanized mathematics. *Journal of Automated Reasoning* 26, pages 269–289, 2001.
- [8] MathScheme : An Integrated Framework for Computer Algebra and Computer Theorem Proving. Home page at <http://imps.mcmaster.ca/mathscheme/>.
- [9] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. Computer aided verification : 8th international conference. *Addison-Wesley*, 1102:411–414, 1996.
- [10] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [11] Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.0*, 2006. Available at <http://coq.inria.fr/doc>.
- [12] S. Wolfram. *Mathematica : A system for doing mathematics by computer*. *Addison-Wesley*, 1991.