McMaster University
DigitalCommons@McMaster

Open Access Dissertations and Theses | Open Dissertations and Theses

4-1-2012

REASONING ABOUT DEFINEDNESS — A DEFINEDNESS CHECKING SYSTEM FOR AN IMPLEMENTED LOGIC

Qian Hu
McMaster University, huq6@mcmaster.ca

Follow this and additional works at: http://digitalcommons.mcmaster.ca/opendissertations

Part of the Computer Sciences Commons, and the Logic and Foundations Commons

Recommended Citation
Hu, Qian, "REASONING ABOUT DEFINEDNESS — A DEFINEDNESS CHECKING SYSTEM FOR AN IMPLEMENTED LOGIC" (2012). *Open Access Dissertations and Theses.* Paper 6435.

This Thesis is brought to you for free and open access by the Open Dissertations and Theses at DigitalCommons@McMaster. It has been accepted for inclusion in Open Access Dissertations and Theses by an authorized administrator of DigitalCommons@McMaster. For more information, please contact scom@mcmaster.ca.

Reasoning about Definedness — A Definedness Checking System for An Implemented Logic

Reasoning about Definedness — A Definedness Checking System for An Implemented Logic

By
Qian Hu, B.Tech.

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 Qian Hu, October 25, 2011

MASTER OF SCIENCE (2011)
(Computer Science)

McMaster University
Hamilton, Ontario

TITLE: Reasoning about Definedness — A Definedness Checking System for an Implemented Logic

AUTHOR: Qian Hu, B.Tech. (University Of Mysore, India)

SUPERVISOR: Dr. William M. Farmer

NUMBER OF PAGES: ix, 102

ABSTRACT

Effective definedness checking is crucial for an implementation of a logic with undefinedness. The objective of the MathScheme project is to develop a new approach to mechanized mathematics that seeks to combine the capabilities of computer algebra systems and computer theorem proving systems. Chiron, the underlying logic of MathScheme, is a logic with undefinedness. Therefore, it is important to automate, to the greatest extent possible, the process of checking the definedness of Chiron expressions for the MathScheme project. This thesis provides an overview of information useful for checking definedness of Chiron expressions and presents the design and implementation of an AND/OR tree-based approach for automated definedness checking based on ideas from artificial intelligence. The theorems for definedness checking are outlined first, and then a three-valued AND/OR tree is presented, and finally, the algorithm for reducing Chiron definedness problems using AND/OR trees is illustrated. An implementation of the definedness checking system is provided that is based on the theorems and algorithm. The ultimate goal of this system is to provide a powerful mechanism to automatically reduce a definedness problem to simpler definedness problems that can be easily, or perhaps automatically, checked.

CONTENTS

Abstract **iii**

1 Introduction **1**

1.1 Questions of Definedness 1

1.2 Definedness Checking in MathScheme 2

1.3 Objectives of the Thesis 3

1.4 Organization of the Thesis 3

1.5 Fonts 4

2 Background **6**

2.1 MathScheme 6

2.2 Chiron 7

2.2.1 Values 7

2.2.2 Expressions 8

2.2.3 Additional Notation 13

2.2.4 Implementation of Chiron Expressions 14

2.3 Simplification and Contexts in MathScheme 16

2.4 Definedness Checking in IMPS 17

3 Problem and Objectives **19**

3.1 Definedness in Chiron 19

3.2 Definedness Checking of Chiron Expressions 20

4 Theorems about Definedness in Chiron **23**
4.1 Overview 23
4.2 Definedness Theorems 23

5 AND/OR Trees **31**
5.1 Overview 31
5.2 Formalization of AND/OR Trees for Definedness Problems 32
5.3 Reducing Definedness Problems in Chiron Using Starter Trees 35
5.3.1 General Cases 36
5.3.2 Operator Application 37
5.3.3 Constants 39
5.3.4 Variables 39
5.3.5 Type Application 40
5.3.6 Dependent Function Types 41
5.3.7 Function Application 43
5.3.8 Function Abstraction 44
5.3.9 Conditional Terms 45
5.3.10 Definite Description 47
5.3.11 Indefinite Description 49
5.3.12 Quotation 50
5.3.13 Evaluation 51

6 Definedness Checking Mechanism **54**
6.1 Overview 54
6.2 Classical AND/OR Tree Search Methods 55
6.2.1 Definitions and Notation 55
6.2.2 Breadth-First Search 56
6.2.3 Depth-First Search 57
6.2.4 Other Search Methods 59
6.3 Definedness Checking Algorithm 59
6.3.1 Overview of Definedness Checking Algorithm 60
6.3.2 Steps of Definedness Checking Algorithm 61

7 Implementation **63**
7.1 Overview 63
7.2 Implementation 64

7.2.1 Building Starter Trees 64
7.2.2 Leaf Checking Procedure 65
7.2.3 Node Labeling Procedure 67
7.3 Expanding Trees 68

8 Conclusion **70**

9 Future Work **72**

Acknowledgements **74**

APPENDIX — Source Code **75**
A.1 Type Definitions for AND/OR Trees 75
A.2 Implementing Categories of Starter Trees 76
A.2.1 General Cases 76
A.2.2 Operator Application 78
A.2.3 Type Application 82
A.2.4 Dependent Function Types 83
A.2.5 Function Application 84
A.2.6 Function Abstraction 86
A.2.7 Conditional Terms 87
A.2.8 Definite Description 90
A.2.9 Indefinite Description 91
A.2.10 Quotation 92
A.2.11 Evaluation 93
A.3 Building Starter Trees 95
A.4 Node Labeling Procedure 97
A.5 Leaf Checking Procedure 98
A.6 Function for Definedness Checking 99

Bibliography **99**

LIST OF FIGURES

5.1 AND/OR tree for $D \vee (B \wedge C)$. 33
5.2 Example of a solution graph of an AND/OR tree. 35
5.3 AND/OR tree for the general cases of being defined in a type. 36
5.4 AND/OR tree for general cases undefined in a type. 37
5.5 AND/OR tree for type operator applications equal to C. 38
5.6 AND/OR tree for undefined term operator application. 38
5.7 AND/OR tree for formula operator application false. 39
5.8 AND/OR tree for empty type applications. 40
5.9 AND/OR tree for type applications equal to C. 41
5.10 AND/OR tree for function is defined in a dependent function type. . 42
5.11 AND/OR tree for functions undefined in a dependent function type. . 42
5.12 AND/OR tree for function application is defined in a given type. . . . 43
5.13 AND/OR tree for function application is undefined in a given type. . 44
5.14 AND/OR tree for function abstraction is defined. 45
5.15 AND/OR tree for conditional term is defined in a given type. 46
5.16 AND/OR tree for conditional term is undefined in a given type. . . . 47
5.17 AND/OR tree for definite description is defined in a given type. . . . 48
5.18 AND/OR tree for definite description is undefined in a given type. . . 48
5.19 AND/OR tree for indefinite description is defined in a given type. . . 49
5.20 AND/OR tree for indefinite description is undefined in a given type. . 50
5.21 AND/OR tree for type quotation is defined in a given type. 51
5.22 AND/OR tree for term quotation is defined in a given type. 51

5.23 AND/OR tree for type evaluation is of C type. 52
5.24 AND/OR tree for term evaluation is undefined. 52
5.25 AND/OR tree for formula evaluation is false. 53

6.1 An AND/OR tree showing the order of node expansions in a breadth-first search. 58
6.2 An AND/OR tree showing the order of node expansions in a depth-first search (depth-bound = 2). 59

LIST OF TABLES

2.1 Chiron notation for proper expressions 12
2.2 Additional notations 13
2.3 Implementation of Chiron expressions 15
2.4 Implementation of variable binders 16

5.1 AND node status truth table 34
5.2 OR node status truth table 34

CHAPTER 1

INTRODUCTION

1.1 Questions of Definedness

In a standard logic all functions are total and all terms denote some value, so partial functions must be represented by total functions and an undefined term must be given a value. But in mathematics and computer science, undefined terms arise naturally. In the following we will give an explanation of definedness, discuss sources of definedness, and describe approaches to dealing with undefinedness based on [6, 7].

A mathematical term is *undefined* if it has no prescribed meaning or if it denotes a value that does not exist. There are three sources of undefinedness:

(1) Improper function application.

(2) Improper definite description.

(3) Improper indefinite description.

A function f has both a domain of definition D_{f} consisting of the values at which it is defined and a domain of application D_{f}^* consisting of the values to which it may be applied. A function f is total if $D_{\mathrm{f}} = D_{\mathrm{f}}^*$ and strictly partial if $D_{\mathrm{f}} \subset D_{\mathrm{f}}^*$. A function application $f(a)$ is undefined if f is undefined, a is undefined, or $a \notin D_{\mathrm{f}}$. An example of improper function application is $\sqrt{-4}$.

A *definite description* is a term t of the form "the x such that A holds". t is undefined if there is no x such that A holds or there is more than one x such that A holds. An *indefinite description* is a term t of the form "some x such that A holds". t is undefined if there is no x such that A holds. Definite and indefinite descriptions naturally lead to undefined terms in mathematics. For example, "the $x \in \mathtt{R}$ such that $x^2 = 4$" is a definite description which is undefined since two values (2 and -2) satisfy the formula $x^2 = 4$; "some $x \in \mathtt{R}$ such $x^2 = -4$" is a undefined indefinite description since no such x satisfies $x^2 = -4$.

There is a traditional approach to dealing with undefinedness that is widely practiced in mathematics based on three principles:

- Atomic terms (i.e., variables and constants) are always defined.
- Compound terms may be undefined.
  (1) An application of a function $f(a)$ is undefined if f is undefined, a is undefined, or $a \notin D_{\mathrm{f}}$.
  (2) A definite description is undefined if there is no x such that A holds or there is more than one x such that A holds.
  (3) An indefinite description is undefined if there is no x such that A holds.
- Formulas are always true or false, and hence, are always defined. A predicate application $p(a)$ is false if p is undefined, a is undefined, or $a \notin \mathrm{dom(p)}$. Predicates must be total.

There are three choices to formalize undefinedness. The first is to formalize partial functions and undefined terms in a standard logic. The second is to select or develop a three-valued logic which admits both undefined terms and undefined formulas. And the last is to select or develop a two-valued logic with undefinedness.

1.2 Definedness Checking in MathScheme

MathScheme [16], which is being developed at the McMaster University, is a project to build a software system to help mathematics practitioners (engineers, scientists, mathematicians, etc.) to do mathematics, especially to do rigorous mathematical

reasoning. It is a new approach to mechanized mathematics, which seeks to combine the capabilities of computer algebra systems such as Axiom [14], Maple [2], and Mathematica [24] and computer theorem proving systems such as Coq [22], Isabelle [20], and PVS [21]. The underlying logic of MathScheme, named Chiron [8, 9], is a two-valued logic with undefinedness. So as part of the MathScheme project, a powerful system is needed that has an intelligent mechanism for checking definedness in Chiron.

1.3 Objectives of the Thesis

The subject of this thesis is the design and code implementation of a definedness checking system for the MathScheme project based on the logic Chiron. This thesis will introduce the concept of logics with undefinedness and explore a definedness checking system as a unit inside a mechanized mathematics system that can perform the task of checking the definedness of terms. The goal is to develop a Chiron definedness checking system for the MathScheme project that automatically reduces a definedness problem to simpler definedness problems that can be easily, or perhaps automatically, checked.

1.4 Organization of the Thesis

This thesis illustrates how to build a Chiron definedness checking system for the MathScheme project. It presents several theorems of definedness and starter AND/OR trees for proper expressions of the Chiron logic. As such, the document is divided into several major chapters, each of which deals with a different solution piece of the Chiron definedness checking system. It ends with an appendix of the source code, which shows the details of the definedness checking system, and a bibliography. The following is the outline of this thesis.

Chapter 2 (Background) presents the background concepts for our Chiron definedness checking system. It aims to give a brief overview of all the material needed for understanding this thesis. A description of MathScheme, Chiron, simplification, and the definedness checking mechanism in IMPS are found in this chapter.

Chapter 3 **(Problem and Objectives)** gives an overview of our research problem and research objectives.

Chapter 4 **(Theorems about Definedness in Chiron)** presents the theorems of definedness for our work. These theorems are organized by the different categories of proper expressions in Chiron.

Chapter 5 **(AND/OR Trees)** discusses AND/OR trees. In this chapter, we introduce the main idea of and AND/OR trees and present how AND/OR trees can represent the different categories of definedness problems in Chiron.

Chapter 6 **(Definedness Checking Mechanism)** revisits classical search methods such as breadth-first and depth-first methods and establishes a novel definedness checking algorithm that is used for implementing the functionalities of the definedness checking system.

Chapter 7 **(Implementation)** provides a description of the implementation for our definedness checking algorithm.

Chapter 8 **(Conclusion)** gives a brief summary of the thesis.

At the end, **Chapter** 9 **(Future Work)** proposes some improvements to our system for future work.

Appendix (Source Code) contains the source code of the implementation of the program.

1.5 Fonts

There are several fonts are used in this thesis for special purposes:

- *Italics* — for identifying a term that is being defined.
- Sans serif — for the names of Chiron symbols such as op-app.

- **Bold** — for the OCaml types that represent sorts of Chiron expressions such as **sexpression**.
- `Typewriter` — for OCaml code such as the function name `node_rec`.

CHAPTER 2

BACKGROUND

2.1 MathScheme

MathScheme is intended to be a new mechanized mathematics system for symbolic computation and mathematical reasoning that is useful to a wide range of people. It combines the strengths of symbolic algebra systems with those of formal deduction systems to yield a powerful and sound system. The mission of the MathScheme project can be found at its homepage [16].

The short-term goals of the MathScheme project are:

(1) To develop a formal framework that integrates symbolic computation and formal deduction.

(2) To design and implement an MMS based on the formal framework. It should allow one to build formal languages, theories, computations, and mappings between theories.

The long-range goal is to build an interactive mathematics laboratory that has the capabilities of both computer algebra systems and computer theorem proving systems and the means to formalize a wide range of mathematical knowledge. It is intended to be a very effective tool for formalizing and reasoning about mathematics.

2.2 Chiron

The logic Chiron, designed and developed by Dr. Farmer, is the logical basis for the MathScheme mechanized mathematics system. The Chiron technique report [9] defines the values and expressions of Chiron and presents a formal definition of the syntax and semantics of Chiron. This section aims to give a brief overview of Chiron and to review the proper expressions of Chiron as well as how these expressions are implemented.

Chiron is a derivative of von-Neumann-Bernays-Gödel set theory that is intended to be a practical, general-purpose logic for mechanizing mathematics. It integrates several reasoning paradigms, namely classical, permitted undefinedness, set theory, type theory and formalized syntax [8]. It has a type system with a universal type, dependent types, dependent function types, subtypes, and possibly empty types.

Unlike traditional logics such as first-order logic and simple type theory, Chiron has a special mechanism for handling undefinedness. Ungrounded terms are treated as being undefined, and ungrounded formulas are treated as being false. It also has a facility for reasoning about the syntax of expressions that employs quotation and evaluation. Chiron is well suited for MathScheme project because of its expressiveness and strong support for dealing with undefinedness and reasoning about syntax.

2.2.1 Values

A *value* is a set, class, superclass, truth value, undefined value, or operation. In Chiron, the domain D_{v} of sets is a proper subdomain of the domain D_{c} of classes, and D_{c} is a proper subdomain of the the domain D_{s} of superclasses. D_{v} is the universal class (the class of all sets) and D_{c} is the universal superclass (the superclass of all classes).

There are two truth values, T representing *true* and F representing *false*, where the truth values are not members of D_{s}. There is also an *undefined value* $\perp$ which serves as the value of various sorts of undefined terms. $\perp$ is not a member of $D_s \cup \{\mathrm{T}, \mathrm{F}\}$.

For $n \geq 0$, an *n-ary operation* is a total mapping

$$\sigma : D_1 \times \cdots \times D_n \to D_{n+1}$$

where D_i is D_{s}, $D_{\mathrm{c}} \cup \{\bot\}$, or $\{\mathrm{T}, \mathrm{F}\}$ for all i with $1 \leq i \leq n+1$. An operation is not a member of $D_s \cup \{\mathrm{T}, \mathrm{F}, \bot\}$.

2.2.2 Expressions

An expression of Chiron is inductively defined by the following two formation rules:

Expr-1 (Atomic expression)

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

Expr-2 (Compound expression)

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

where $n \geq 0$.

In these formation rules, $\mathcal{O}$ is a countable set of symbols called operator names, $\mathcal{S}$ is a fixed countably infinite set of Chiron symbols and $\mathbf{expr}_e[]$ asserts that e is an expression of L. Hence, an expression is the same as an S-expression in Lisp.

There are four special sorts of expressions: *operators*, *types*, *terms*, and *formulas*. A *proper expression* is one of these special sorts of expressions. *Improper expressions* are expressions that are not proper expressions. Proper expressions denote classes, super classes (i.e., a sets of classes), truth values, the undefined value and operations, while improper expressions are nondenoting. In Chiron, operators denote operations; types, which are used to restrict the values of operators and variables and to classify terms by their values, denote superclasses; terms denote classes or the undefined value $\bot$; formulas denote truth values. A term is *defined* if it denotes a class and is *undefined* if it denotes $\bot$.

The following formation rules is presented to define the 25 proper expression categories:

P-Expr-1 (Operator)

$$\frac{o \in \mathcal{O}, \mathbf{kind}_L[k_1], \ldots, \mathbf{kind}_L[k_{n+1}]}{\mathbf{operator}_L[(\mathsf{op}, o, k_1, \ldots, k_{n+1})]}$$

where $n \geq 0$; $\theta(o) = s_1, \ldots, s_{n+1}$; and $k_i = s_i = \mathsf{type}$, $\mathbf{type}_L[k_i]$ and $s_i = \mathsf{term}$, or $k_i = s_i = \mathsf{formula}$ for all i with $1 \leq i \leq n+1$.

P-Expr-2 (Operator application)

$$\frac{\mathbf{operator}_L[(\mathsf{op}, o, k_1, \ldots, k_{n+1})], \mathbf{expr}_L[e_1], \ldots, \mathbf{expr}_L[e_n]}{\mathbf{p\text{-}expr}_L[(\mathsf{op\text{-}app}, (\mathsf{op}, o, k_1, \ldots, k_{n+1}), e_1, \ldots, e_n) : k_{n+1}]}$$

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

P-Expr-3 (Constant)

$$\frac{\mathbf{operator}_L[(\mathsf{op}, o, k)]}{\mathbf{p\text{-}expr}_L[(\mathsf{con}, o, k) : k]}$$

P-Expr-4 (Variable)

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

P-Expr-5 (Type application)

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

P-Expr-6 (Dependent function type)

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

P-Expr-7 (Function application)

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

P-Expr-8 (Function abstraction)

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

P-Expr-9 (Conditional term)

$$\frac{\mathbf{formula}_L[A], \mathbf{term}_L[b : \beta], \mathbf{term}_L[c : \gamma]}{\mathbf{term}_L[(\mathsf{if}, A, b, c) : \beta \cup \gamma]}$$

P-Expr-10 (Existential quantification)

$$\frac{\mathbf{term}_L[(\mathsf{var}, x, \alpha)], \mathbf{formula}_L[B]}{\mathbf{formula}_L[(\mathsf{exists}, (\mathsf{var}, x, \alpha), B)]}$$

P-Expr-11 (Unique existential quantification)

$$\frac{\mathbf{term}_L[(\mathsf{var}, x, \alpha)], \mathbf{formula}_L[B]}{\mathbf{formula}_L[(\mathsf{uni\text{-}exists}, (\mathsf{var}, x, \alpha), B)]}$$

P-Expr-12 (Universal quantification)

$$\frac{\mathbf{term}_L[(\mathsf{var}, x, \alpha)], \mathbf{formula}_L[B]}{\mathbf{formula}_L[(\mathsf{forall}, (\mathsf{var}, x, \alpha), B)]}$$

P-Expr-13 (Definite description)

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

P-Expr-14 (Indefinite description)

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

P-Expr-15 (Set construction)

$$\frac{\mathbf{term}_L[a_1 : \alpha_1], \ldots, \mathbf{term}_L[a_1 : \alpha_1]}{\mathbf{term}_L[(\mathsf{set\text{-}cons}, a_1, \ldots, a_n) : \beta]}$$

where $n \geq 0$ and $\beta = \begin{cases} \mathsf{C} & \text{if } n = 0 \\ \alpha_1 \cup \cdots \cup \alpha_n & \text{otherwise.} \end{cases}$

P-Expr-16 (List construction)

$$\frac{\mathbf{term}_L[a_1 : \alpha_1], \ldots, \mathbf{term}_L[a_1 : \alpha_1]}{\mathbf{term}_L[(\mathsf{list\text{-}cons}, a_1, \ldots, a_n) : \mathsf{list\text{-}type}(\beta)]}$$

where $n \geq 0$ and $\beta = \begin{cases} \mathsf{C} & \text{if } n = 0 \\ \alpha_1 \cup \cdots \cup \alpha_n & \text{otherwise.} \end{cases}$

P-Expr-17 (Class abstraction)

$$\frac{\mathbf{term}_L[(\mathsf{var}, x, \alpha)], \mathbf{formula}_L[B]}{\mathbf{term}_L[(\mathsf{class\text{-}abs}, (\mathsf{var}, x, \alpha), B) : \mathsf{power\text{-}type}(\alpha)]}$$

P-Expr-18 (Left type)

$$\frac{\mathbf{type}_L[\alpha]}{\mathbf{type}_L[(\mathsf{left\text{-}type}, \alpha)]}$$

P-Expr-19 (Right type)

$$\frac{\mathbf{type}_L[\alpha], \mathbf{term}_L[a]}{\mathbf{type}_L[(\mathsf{right\text{-}type}, \alpha, a)]}$$

P-Expr-20 (Dependent type product)

$$\frac{\mathbf{term}_L[(\mathsf{var}, x, \alpha)], \mathbf{type}_L[\beta]}{\mathbf{type}_L[(\mathsf{dep\text{-}type\text{-}prod}, (\mathsf{var}, x, \alpha), \beta)]}$$

P-Expr-21 (Dependent ordered pair)

$$\frac{\mathbf{term}_L[a : \alpha], \mathbf{term}_L[b : \beta]}{\mathbf{term}_L[(\mathsf{dep\text{-}ord\text{-}pair}, a, b) : (\mathsf{dep\text{-}type\text{-}prod}, (\mathsf{var}, x, \alpha), \beta)]}$$

P-Expr-22 (Dependent head)

$$\frac{\mathbf{term}_L[a : \alpha]}{\mathbf{term}_L[(\mathsf{dep\text{-}head}, a) : (\mathsf{left\text{-}type}, \alpha)]}$$

P-Expr-23 (Dependent tail)

$$\frac{\mathbf{term}_L[a : \alpha]}{\mathbf{term}_L[(\mathsf{dep\text{-}tail}, a) : (\mathsf{right\text{-}type}, \alpha, (\mathsf{dep\text{-}head}, a))]}$$

P-Expr-24 (Quotation)

$$\frac{\mathbf{expr}_L[e]}{\mathbf{term}_L[(\mathsf{quote}, e) : \alpha]}$$

where α is:

(1) E_{sy} if $e \in \mathcal{S}$.

(2) E_{on} if $e \in \mathcal{O}$.

(3) E_{op} if e is an operator.

(4) E_{ty} if e is a type.

(5) $\mathsf{E}_{\mathrm{te}}^{\ulcorner\beta\urcorner}$ if e is a term of type β.

(6) E_{fo} if e is a formula.

(7) E if none of the above.

P-Expr-25 (Evaluation)

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

The official notation and compact notation for those proper expressions are given in Table 2.1. The column "INCLUDED?" shows whether those proper expressions can be checked by our definedness checking system.

Table 2.1: Chiron notation for proper expressions

OFFICIAL NOTATION	COMPACT NOTATION	INCLUDED?
$(\mathsf{op}, o, k_1, \ldots, k_{n+1})$	$(o :: k_1, \ldots, k_{n+1})$	NO
$(\mathsf{op\text{-}app}, (\mathsf{op}, o, k_1, \ldots, k_{n+1}), e_1, \ldots, e_n)$	$(o :: k_1, \ldots, k_{n+1})(e_1, \ldots, e_n)$	YES
(con, o, k)	$[o :: k]$	YES
$(\mathsf{var}, x, \alpha)$	$(x : \alpha)$	YES
$(\mathsf{type\text{-}app}, \alpha, a)$	$\alpha(a)$	YES
$(\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$	$(\Lambda\, x : \alpha \,.\, \beta)$	YES
$(\mathsf{fun\text{-}app}, f, a)$	$f(a)$	YES
$(\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$	$(\lambda\, x : \alpha \,.\, b)$	YES
(if, A, b, c)	$\mathsf{if}(A, b, c)$	YES
$(\mathsf{exists}, (\mathsf{var}, x, \alpha), B)$	$(\exists\, x : \alpha \,.\, B)$	NO
$(\mathsf{uni\text{-}exists}, (\mathsf{var}, x, \alpha), B)$	$(\exists!\, x : \alpha \,.\, B)$	NO
$(\mathsf{forall}, (\mathsf{var}, x, \alpha), B)$	$(\forall\, x : \alpha \,.\, B)$	NO
$(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$	$(\iota\, x : \alpha \,.\, B)$	YES
$(\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$	$(\epsilon\, x : \alpha \,.\, B)$	YES
$(\mathsf{set\text{-}cons}, a_1, \ldots, a_n)$	$\{a_1, \ldots, a_n\}$	NO
$(\mathsf{list\text{-}cons}, a_1, \ldots, a_n)$	$[a_1, \ldots, a_n]$	NO
$(\mathsf{class\text{-}abs}, (\mathsf{var}, x, \alpha), B)$	$(\mathrm{C}\, x : \alpha \,.\, B)$	NO
$(\mathsf{left\text{-}type}, \alpha)$	$\mathsf{left}(\alpha)$	NO

$(\mathsf{right\text{-}type}, \alpha, a)$	$\mathsf{right}(\alpha, a)$	NO
$(\mathsf{dep\text{-}type\text{-}prod}, (\mathsf{var}, x, \alpha), \beta)$	$(x : \alpha \,.\, \beta)$	NO
$(\mathsf{dep\text{-}ord\text{-}pair}, a, b)$	$\langle a, b \rangle$	NO
$(\mathsf{dep\text{-}head}, a)$	$\mathsf{hd}(a, b)$	NO
$(\mathsf{dep\text{-}tail}, a)$	$\mathsf{tl}(a, b)$	NO
(quote, e)	$\ulcorner e \urcorner$	YES
$(\mathsf{quasiquote}, (\mathsf{fun\text{-}app}, f, \lfloor a \rfloor))$	$\ulcorner f(\lfloor a \rfloor) \urcorner$	NO
(eval, a, k)	$[\![a]\!]_k$	YES

Because the official notation shows the exact structures of expressions, we use it to give the definition of Chiron expressions in the rest of this thesis. When we describe those expressions, such as in building starter trees, we mainly use the compact notation instead of the official notation for convenience.

2.2.3 Additional Notation

The notational definitions in Table 2.2 defines additional compact notation used in this thesis. The first ten notations denote types; the row 11-14 denote equalities; the row 15-20 denote logical operators; the row 21-25 denote definedness operators; the row 26-31 denote relation of types; the rest of the table denote additional build-in operators.

Table 2.2: Additional notations

COMPACT NOTATION	OFFICIAL NOTATION
V	$(\mathsf{con}, \mathsf{set}, \mathsf{type})$
C	$(\mathsf{con}, \mathsf{class}, \mathsf{type})$
E	$(\mathsf{con}, \mathsf{expr}, \mathsf{type})$
E_{sy}	$(\mathsf{con}, \mathsf{expr\text{-}sym}, \mathsf{type})$
E_{on}	$(\mathsf{con}, \mathsf{expr\text{-}op\text{-}name}, \mathsf{type})$
E_{op}	$(\mathsf{con}, \mathsf{expr\text{-}op}, \mathsf{type})$
E_{ty}	$(\mathsf{con}, \mathsf{expr\text{-}type}, \mathsf{type})$
E_{te}	$(\mathsf{con}, \mathsf{expr\text{-}term}, \mathsf{type})$

$\mathsf{E}^{a}_{\mathrm{te}}$	(op-app, (op, expr-term-type, E_{ty}, type), a)
E_{fo}	(con, expr-formula, type)
$(\alpha =_{\mathrm{ty}} \beta)$	(op-app, (op, type-equal, type, type, formula), α, β)
$(a = b)$	(op-app, (op, term-equal, C, C, type, formula), a, b, C)
$(a \simeq b)$	(op-app, (op, quasi-equal, C, C, formula), a, b)
$(A \equiv B)$	(op-app, (op, formula-equal, formula, formula, formula), A, B)
T	(con, true, formula)
F	(con, false, formula)
$(\neg A)$	(op-app, (op, not, formula, formula), A)
$(A \vee B)$	(op-app, (op, or, formula, formula, formula), A, B)
$(A \wedge B)$	(op-app, (op, and, formula, formula, formula), A, B)
$(A \supset B)$	(op-app, (op, implies, formula, formula, formula), A, B)
$(a \downarrow \alpha)$	(op-app, (op, defined-in, C, type, formula), a, α)
$(a\downarrow)$	(op-app, (op, defined-in, C, type, formula), a, C)
$(a \uparrow \alpha)$	(op-app, (op, undefined-in, C, type, formula), a, α)
$(a\uparrow)$	(op-app, (op, undefined-in, C, type, formula), a, C)
$\perp_{\mathsf{C}}$	(con, undefined, C)
∇	(con, empty-type, type)
$(\alpha \ll \beta)$	(op-app, (op, type-le, type, type, formula), α, β)
$(\alpha \rightarrow \beta)$	(op-app, (op, sim-fun-type, type, type, type), α, β)
$(a \in b)$	(op-app, (op, in, V, C, formula), a, b)
$(\alpha \cap \beta)$	(op-app, (op, type-intersection, type, type, type), α, β)
$\emptyset$	(con, empty-set, V)
$a \subseteq b$	(op-app, (op, subclass, C, C, formula), a, b)
term(α)	(op-app, (op, type-to-term, type, C), α)
type(a)	(op-app, (op, term-to-type, C, type), α)
total(f, term(α))	(op-app, (op, total, C, C, formula), f, x)
gea($\ulcorner e_1 \urcorner$, $\ulcorner e_2 \urcorner$)	(op-app, (op, gea, E, E, formula), $\ulcorner e_1 \urcorner$, $\ulcorner e_2 \urcorner$)
strict(o)	(op-app, (op, is-strict-op-name, E_{on}, formula), o)

2.2.4 Implementation of Chiron Expressions

Ni Hong [18] used the type system of Objective Caml (OCaml) [1] to implement the set of Chiron expressions. In the implementation, an expression type is defined as

a subtype of another expression type. The `Types` and `Keywords` modules establish the fundamental type system of Chiron and offer tools for building Chiron expressions. The `Keywords` module defines the key words of Chiron and the `Types` module formalizes all Chiron expressions including proper expressions and improper expressions as S-expressions by using polymorphic variant types. There are three variable binder types, **tbinder**, **fbinder**, and **binder**. Proper expressions are organized into four OCaml types **operator**, **ctype**, **term**, and **formula**. Table 2.3 shows the corresponding expressions of four proper expression types. Table 2.4 shows the constructors of three variable binders.

Table 2.3: Implementation of Chiron expressions

Data Type	Constructor	Description
operator	`Operator`	operator
ctype	`TConstant`	type constant
	`TOpApp`	type operator application
	`TTypeApp`	type application
	`TBinder`	type binder
	`TEval`	type evaluation
term	`‘Constant`	term constant
	`‘OpApp`	term operator application
	`‘Var`	variable
	`‘FunApp`	function application
	`‘FunAbs`	function abstraction
	`‘If`	conditional term
	`‘Binder`	term binder
	`‘Quote`	quotation
	`‘Eval`	term evaluation
formula	`FConstant`	formula constant
	`FOpApp`	formula operator application
	`FBinder`	formula binder
	`FEval`	formula evaluation

Table 2.4: Implementation of variable binders

Data Type	Constructor	Description
tbinder	`Dep_fun_type`	dependent function type
	`Dep_type_prod`	dependent type product
binder	`Def_des`	definite description
	`Indef_des`	indefinite description
	`Class_abs`	Class abstraction
fbinder	`Exists`	existential quantification
	`Uni_exists`	unique existential quantification
	`Forall`	universal quantification

2.3 Simplification and Contexts in MathScheme

A simplifier is an important part in many proof assistants. The simplifier for MathScheme is designed to be a tool that provide facilities to help the end users or other processes of the system accomplish their proofs and computation tasks. As a component of MathScheme project, Han Yin Zhang presented an implementation of a framework for simplifying Chiron expressions as part of his master's research [25].

In the MathScheme framework, a simplifier is a transformer which maps an input expression to a simpler expression, where both expressions must denote the same semantic value. The major routine of the simplification is repeated application of simplification rules to a given expression. One kind of simplification rule is a rewrite rule which is represented by an equation such as $x + x + x = 3 \times x$. By applying this rule, we can simplify expression $2 + 2 + 2$ to 3×2. The other kind of rules are designed to deal with complex tasks which cannot be represented by equations. The simplification result also depends on the context, i.e., the background assumptions.

The definition of context and some context-based techniques for symbolic computation have been given in [12]. A *context* is defined as a set of formulas $\Gamma = \{_1, \ldots, _n\}$. Those formulas ordinarily serve as background assumptions. A formula B is true in the context if the members of the context logically imply B. A global context contains

all the axioms and mathematical knowledge we know. Whereas a "context at a place in a formula" is called a *local context* [17]. In rigorous mathematical reasoning, contexts are necessary to manage the assumptions that arise in the course of reasoning and provide a mechanism for keeping track of what can be assumed at various places in a formula.

In the implementation of the simplifier for MathScheme project, when the simplifier is given to a formula F, it first checks if this formula is in the context. If F is in the context, simplifier will immediately simplify F to T. If the negation of F is in the context, simplifier will simplify F to F. Otherwise, simplifier will call our definedness checking system to check the definedness of F.

2.4 Definedness Checking in IMPS

IMPS, an Interactive Mathematical Proof System [3, 5, 11], is a mathematical proof system developed in the early 1990s at the MITRE Corporation. The IMPS system has proven to be a very effective and powerful proof system for formalizing and reasoning about traditional mathematics. A major part of the success of IMPS is due to its ability to effectively deal with partial functions and undefined terms.

LUTINS [4, 5, 13], a version of simple type theory that admits partial functions, undefined terms, and subtypes, is the logical basis of IMPS. Because LUTINS does not assume that all functions are total and all expressions are defined, term simplification in LUTINS must check the definedness of expressions in the course of proof. The algorithm in IMPS for definedness checking is embedded in the IMPS simplifier, which separates the simplifying of definedness assertions into two levels (lower level and upper level), according to whether recursive calls to the simplifier are involved [12, 10]. The two levels of the definedness checking algorithm are introduced in the following paragraphs.

In the lower level of definedness checking, there are no recursive calls to the simplifier. There are two theorems used in the lower level:

- *Totality*:

$$\forall x :_1, \ldots, x_n : \alpha_n \,.\, f(x_1, \ldots, x_n) \downarrow \alpha$$

where $i = 1, \ldots, n$

- *Unconditional Sort coercions*:

$$\forall x : \alpha \,.\, x \downarrow \beta$$

The relation of sorts together with the totality theorems are used in IMPS to check the definedness. The results of the lower level of definedness checking are passed to the upper layer, which uses conditional information.

In the upper level, the simplifier attempts to reduce the resulting assertions to truth. There are two primary kinds of information about the domain and range of functions, and the relations between sorts, in the theory of this level.

- *Definedness conditions* of the form:

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, \psi(x_1, \ldots, x_n) \supset f(x_1, \ldots, x_n) \downarrow \alpha$$

- *Value information* of the form:

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, \phi(x_1, \ldots, x_n, g(x_1, \ldots, x_n)) \downarrow \alpha$$

- *Conditional sort coercions*:

$$\forall x : \beta \,.\, \phi(x) \supset x \downarrow \alpha$$

We will use similar ideas of IMPS for our work. The Chiron definedness checking system will check definedness in a type and employ theorems concerning the domain and range of functions and the relations between types. And it will also use the method of local context to check definedness within an expression.

CHAPTER 3

PROBLEM AND OBJECTIVES

3.1 Definedness in Chiron

Because Chiron supports the paradigm of permitting undefinedness, in which terms may be undefined and functions may be partial, many questions about the definedness of expressions must be answered in the course of a proof.

The undefined expressions are handled in Chiron according to the traditional approach [6, 7] to undefinedness:

- The value of an undefined term is the undefined value $\perp$.
- The value of an undefined type is the universal superclass D_{c}.
- The value of an undefined formula is F.

Definedness is expressed in Chiron by two operators: $e \downarrow \alpha$ and $e\downarrow$. The first read "e is defined in α", means the e is defined and its value is in the class denoted by α. The latter is read "e is defined", which means e is defined and its value is of the type denoted by C.

3.2 Definedness Checking of Chiron Expressions

If a system is developed based on the Chiron logic such as MathScheme, it must provide automated support for checking that expressions are well defined or defined with a value in a particular type.

Now we think about an example from [6]:

$$\forall\, x, y : \mathbb{Z}, z : \mathbb{Q} \,.\, 2 < z \supset (x * y - 3! \div z) \downarrow \mathbb{Q}$$

To prove this assertion in MathScheme, we need to check some definedness facts including:

- $\mathbb{N}$ is nonempty.
- $\mathbb{Z}$ is nonempty.
- $\mathbb{Q}$ is nonempty.
- $\mathbb{N}$ is a subtype of $\mathbb{Z}$.
- $\mathbb{Z}$ is a subtype of $\mathbb{Q}$.
- $3 \downarrow \mathbb{N}$.
- ! is defined at 3.
- $*$ and $-$ are total on $\mathbb{Z} \times \mathbb{Z}$.
- $\forall\, x, y : \mathbb{Z} \,.\, y \neq 0 \supset x \div y \downarrow \mathbb{Z}$.
- $\forall\, x : \mathbb{Q} \,.\, 2 < x \supset x \neq 0$.

This thesis presents a Chiron definedness checking system to check the definedness problems of formulas like those above. It provides automated support for checking the definedness of Chiron expressions in the course of a proof, so that users can avoid proving a large number of (mostly trivial) theorems. The ultimate goal of this system is to provide a powerful mechanism to automatically reduce a definedness problem to simpler definedness problems that can be easily, or perhaps automatically, checked. The objectives of our system are listed as follows.

Objective 1: To provide (constructive) theorems for checking the definedness of Chiron expressions.

There are several possibilities of undefined expressions in Chiron:

- Operator, type, and function applications may be nondenoting.
- Function abstractions may be nonexistent.
- Definite and indefinite descriptions may be improper.
- Out of range variables and evaluations.

The definedness checking system should have a collection of theorems for determining whether an expression is defined or not and concerning the relations between types.

Objective 2: To design a representation of definedness problems.

A definedness problem can often be reduced to other definedness problems. An AND/OR tree is a good data structure for representation the reduction of definedness problems. By using AND/OR trees, it is easy to represent definedness problems and reduce them to simpler problems which can be checked by definedness checking rules. The definedness checking system should have the following categories of AND/OR trees based on the main categories of proper expressions of Chiron:

- Operator and operator applications.
- Constants and variables.
- Type applications and dependent function types.
- Function applications and abstractions.
- Conditional terms.
- Definite and indefinite descriptions.
- Quotations.
- Evaluations.

The other categories of proper expressions, which are not considered in this thesis, can be handled similarly.

Objective 3: To provide a mechanism for performing definedness checking tasks on formulas automatically.

Since the AND/OR trees may be very large, one job of the definedness checking mechanism is to avoid full development of the AND/OR tree implied by the problem. Therefore, as the tree grows, the definedness checking system must make decisions about which leaf nodes to reduce.

We analyse the original definedness problem and simplify it to some subproblems based on the AND/OR trees so that by checking the definedness of the subproblems we can generate a solution to the original problem. In artificial intelligence research, there are many search methods such as breadth-first, depth-first, and best-first methods to find possible solutions. Different problem solving methods have their own benefits when they are used to handle different definedness problems. We should employ a similar problem solving methods to present a novel algorithm for our definedness checking system. This algorithm is capable of reducing definedness problems to simpler problems and checking the definedness conditions easily, and efficiently, and in some cases, fully automatically.

CHAPTER 4

THEOREMS ABOUT DEFINEDNESS IN CHIRON

4.1 Overview

The theorems of definedness are the foundation of the definedness checking system. There is a variety of theory-specific information about Chiron used by the definedness checking system :

- The definedness facts about particular kinds of terms.
- The types of expressions, particularly the types of variables and constants.
- The relationships between types.
- Facts about the domain and range of functions and operators.

This Chapter presents the theorems for definedness checking of the logic Chiron based on the axioms and valuation functions from [9].

4.2 Definedness Theorems

The following theorems specify the definedness facts of equalities.

Theorem 1 (Equivalence Relations)

(1) $=_{\mathrm{ty}}$ *is an equivalence relation.*

(2) $\simeq$ *is an equivalence relation.*

(3) $a \simeq b \equiv (a{\downarrow} \vee b{\downarrow}) \supset a = b$.

Proof Parts 1 and 2 follows from the Axiom Schema 2 in [9]. And part 3 is the definition of quasi-equality constructor $\simeq$. $\square$

The following theorems deal with the definedness of variables.

Theorem 2 (Variable Definedness)

(1) $(x : \alpha){\downarrow} \supset (x : \alpha) \downarrow \alpha$.

(2) $(x : \alpha){\downarrow} \supset (\alpha \neq_{\mathrm{ty}} \nabla)$.

Proof Part 1 from Axiom Schema 5 in [9]. And part 2 follows from the definition of operator $\mathsf{defined\text{-}in}$ and the definition of the valuation function on variables. $\square$

The following theorems specify the general definedness laws for operators. For a kind k, let

$$\overline{k} = \begin{cases} k & \text{if } k = \mathsf{type} \\ \mathsf{C} & \text{if } k \textit{ is a } \mathsf{type} \\ k & \text{if } k = \mathsf{formula}. \end{cases}$$

Theorem 3 (Operator Application)

(1) $e_{i_1} \downarrow k_{i_1} \wedge \cdots \wedge e_{i_m} \downarrow k_{i_m} \supset$
$(o :: k_1, \ldots, k_n, \mathsf{type})(e_1, \ldots, e_n) =_{\mathrm{ty}}$
$(o :: \overline{k_1}, \ldots, \overline{k_n}, \mathsf{type})(e_1, \ldots, e_n)$

where $n \geq 1$ *and* $k_{i_1}, \ldots, k_{i_m}$ *is a subsequence of types in the sequence* $k_1, \ldots, k_n$ *of kinds.*

(2) $e_{i_1} \downarrow k_{i_1} \wedge \cdots \wedge e_{i_m} \downarrow k_{i_m} \supset$
$(o :: k_1, \ldots, k_n, \beta)(e_1, \ldots, e_n) \simeq$
$(o :: \overline{k_1}, \ldots, \overline{k_n}, \beta)(e_1, \ldots, e_n)$

where $n \geq 1$ *and* $k_{i_1}, \ldots, k_{i_m}$ *is a subsequence of types in the sequence* $k_1, \ldots, k_n$ *of kinds.*

(3) $e_{i_1} \downarrow k_{i_1} \wedge \cdots \wedge e_{i_m} \downarrow k_{i_m} \supset$
$(o :: k_1, \ldots, k_n, \mathsf{formula})(e_1, \ldots, e_n) \equiv$
$(o :: \overline{k_1}, \ldots, \overline{k_n}, \mathsf{formula})(e_1, \ldots, e_n)$

where $n \geq 1$ *and* $k_{i_1}, \ldots, k_{i_m}$ *is a subsequence of types in the sequence* $k_1, \ldots, k_n$ *of kinds.*

(4) $(o :: k_1, \ldots, k_n, \beta)(e_1, \ldots, e_n)\downarrow \supset$
$(o :: k_1, \ldots, k_n, \beta)(e_1, \ldots, e_n) =$
$(o :: k_1, \ldots, k_n, \mathsf{C})(e_1, \ldots, e_n)$

where $n \geq 0$.

(5) $(\mathsf{strict}(o) \wedge a\uparrow) \supset$
$(o :: k_1, \ldots, k_{i-1}, \alpha, k_{i+1}, \ldots, k_n, \mathsf{type})$
$(e_1, \ldots, e_{i-1}, a, e_{i+1}, \ldots, e_n) =_{\mathrm{ty}} \mathsf{C}$

where $n \geq 1$.

(6) $(a\downarrow \wedge a \uparrow \alpha) \supset$
$(o :: k_1, \ldots, k_{i-1}, \alpha, k_{i+1}, \ldots, k_n, \mathsf{type})$
$(e_1, \ldots, e_{i-1}, a, e_{i+1}, \ldots, e_n) =_{\mathrm{ty}} \mathsf{C}$

where $n \geq 1$.

(7) $(\mathsf{strict}(o) \wedge a\uparrow) \supset$
$(o :: k_1, \ldots, k_{i-1}, \alpha, k_{i+1}, \ldots, k_n, \beta)$
$(e_1, \ldots, e_{i-1}, a, e_{i+1}, \ldots, e_n)\uparrow$

where $n \geq 1$.

(8) $(a\downarrow \wedge a \uparrow \alpha) \supset$
$(o :: k_1, \ldots, k_{i-1}, \alpha, k_{i+1}, \ldots, k_n, \beta)$
$(e_1, \ldots, e_{i-1}, a, e_{i+1}, \ldots, e_n)\uparrow$

where $n \geq 1$.

(9) $(\mathsf{strict}(o) \wedge a\uparrow) \supset$
$(o :: k_1, \ldots, k_{i-1}, \alpha, k_{i+1}, \ldots, k_n, \mathsf{formula})$

$(e_1, \ldots, e_{i-1}, a, e_{i+1}, \ldots, e_n) \equiv \mathsf{F}$

where $n \geq 1$.

(10) $(a\downarrow \wedge a \uparrow \alpha) \supset$
$(o :: k_1, \ldots, k_{i-1}, \alpha, k_{i+1}, \ldots, k_n, \mathsf{formula})$
$(e_1, \ldots, e_{i-1}, a, e_{i+1}, \ldots, e_n) \equiv \mathsf{F}$

where $n \geq 1$.

(11) $(o :: k_1, \ldots, k_n, \beta)(e_1, \ldots, e_n)\downarrow \supset$
$(o :: k_1, \ldots, k_n, \beta)(e_1, \ldots, e_n) \downarrow \beta$

where $n \geq 0$.

Proof Parts $1-4$, $6, 8, 10-11$ are parts of Axiom Schemas 3. Parts $5, 7, 9$ follow from the definition of the operator is-strict-op-name. □

The definedness laws for the built-in operators named expr-term-type, in, and term-equal are specified by the following theorem.

Theorem 4 (Built-In Operator Definedness)

(1) $a \uparrow \mathsf{E}_{\mathrm{ty}} \supset \mathsf{E}_a =_{\mathrm{ty}} \mathsf{C}$.

(2) $a \in b \supset (a\downarrow \wedge b\downarrow)$.

(3) $a =_\alpha b \supset (a \downarrow \alpha \wedge b \downarrow \alpha)$.

Proof Follows from Axiom Schemas 4. □

The following theorems deal with the extensionality of types.

Theorem 5 (Types)

(1) $\ll$ *is a partial order for types.*

(2) ∇ *is the minimum value in* $\ll$.

(3) C *is the maximum value in* $\ll$.

(4) $\alpha \ll \beta \equiv (\forall x : \mathsf{C} \,.\, x \downarrow \alpha \supset x \downarrow \beta)$.

Proof Part 1 and 4 follow from the definition of operator type-le. Part 2 follows from the definition of operator empty-type, and part 3 follows of the specification of C. □

The following theorems specify the definedness in a type operator.

Theorem 6 (Definedness in a Type)

(1) $(a \downarrow \alpha \wedge \alpha \ll \beta) \supset a \downarrow \beta$.

(2) $(a \downarrow \alpha \wedge \alpha \cap \beta =_{\mathrm{ty}} \nabla) \supset a \uparrow \beta$.

(3) $\forall a : \alpha \,.\, a\downarrow \supset a \downarrow \alpha$.

(4) $a\downarrow \supset a \downarrow \mathsf{C}$.

(5) $a\uparrow \supset a \uparrow \alpha$

where α is of any type.

(6) $a \uparrow \nabla$.

Proof Pick a model $M = (S, V)$, and assume $V_\varphi((a \downarrow) \wedge (\alpha \ll \beta)) \equiv \mathsf{T}$. By the definition of defined-in, $V_\varphi(a) \in V_\varphi(\alpha)$ and $V_\varphi(\alpha) \subseteq V_\varphi(\beta)$ follows from the definition of type-le. Therefore, $V_\varphi(a) \in V_\varphi(\beta)$, and so part 1 is proved to be true. Part 2 can be proved by same method. And the last four parts follow form the definition of operator defined-in. □

The following theorems deal with the definedness of type applications.

Theorem 7 (Type Application)

(1) $\alpha =_{\mathrm{ty}} \nabla \supset \alpha(a) =_{\mathrm{ty}} \nabla$.

(2) $a\uparrow \supset \alpha(a) =_{\mathrm{ty}} \mathsf{C}$.

Proof Pick a model $M = (S, V)$, and assume $V_\varphi(\alpha =_{\mathrm{ty}} \nabla) \equiv \mathsf{T}$. By the definition of type-equal and empty-type, $V_\varphi(\alpha) = \emptyset$. Therefore, $V_\varphi(\alpha)[V_\varphi(a)] = \emptyset$, and so part 1 is proved to be true. Part 2 follows from the definition in [9] of the valuation function on type applications. □

Definedness involving dependent function types is handled by the following theorem.

Theorem 8 (Dependent Function Type)

(1) $(\Lambda\, x : \alpha\ .\ \beta) \neq \nabla$.

(2) $(\alpha' \ll \alpha) \wedge (\forall x : \alpha'\ .\ \beta' \ll \beta) \supset \Lambda\, x : \alpha'\ .\ \beta' \ll \Lambda\, x : \alpha\ .\ \beta$.

Proof

Part 1 $(\lambda\, x : \nabla\ .\ \emptyset) \downarrow (\Lambda\, x : \alpha\ .\ \beta)$ follows from the definition of the valuation function on dependent function types. Therefore, part 1 is proved to be true.

Part 2 Pick a model $M = (S, V)$, and assume $V_\varphi((\alpha' \ll \alpha) \wedge (\forall x : \alpha'\ .\ \beta' \ll \beta) \equiv \mathsf{T})$. Let $f \in V_\phi(\Lambda\, x : \alpha'\ .\ \beta')$. If $f(d)$ is defined, then $d \in V_\phi(\alpha')$ and so $d \in V_\phi(\alpha)$ by hypothesis. Also, when $f(d)$ is defined, $f(d) \in V_{\phi[x \mapsto d]}(\beta')$ and $f(d) \in V_{\phi[x \mapsto d]}(\beta)$ by hypothesis. Therefore, $f \in V_\phi(\Lambda\, x : \alpha\ .\ \beta)$ and so part 2 is proved to be true. □

The laws for definedness of conditional terms are specified by the following two theorem.

Theorem 9 (Conditional Terms)

(1) $A \supset \mathsf{if}(A, b, c) \simeq b$.

(2) $\neg A \supset \mathsf{if}(A, b, c) \simeq c$.

Proof This is Axiom Schemas 8 in [9]. □

The definedness of definite description is handled by the following theorem, one part for proper definite descriptions and another for improper definite descriptions.

Theorem 10 (Definite Description)

(1) $(\exists!\, x : \alpha\ .\ A) \supset (\iota\, x : \alpha\ .\ A) \downarrow \alpha$.

(2) $\neg(\exists!\, x : \alpha\ .\ A) \supset (\iota\, x : \alpha\ .\ A) \uparrow$.

Proof This is Axiom Schemas 9 in [9]. □

The definedness of indefinite description is handled by the following theorem, one part for proper indefinite descriptions and another for improper indefinite descriptions.

Theorem 11 (Indefinite Description)

(1) $(\exists x : \alpha \,.\, A) \supset (\epsilon x : \alpha \,.\, A) \downarrow \alpha)$.

(2) $\neg(\exists x : \alpha \,.\, A) \supset (\epsilon x : \alpha \,.\, A)\uparrow$.

Proof This is the first two parts of Axiom Schema 10 in [9]. □

The following theorem deals with the definedness of quotation.

Theorem 12 (Quotation)

(1) $\ulcorner \alpha \urcorner \downarrow \mathsf{E}_{\mathrm{ty}}$

where α is a type.

(2) $\ulcorner a \urcorner \downarrow \mathsf{E}_{\mathrm{te}}$

where a is a term.

(3) $\ulcorner a \urcorner \downarrow \mathsf{E}_{\mathrm{te}}^{\ulcorner \alpha \urcorner}$

where a is a term of type α.

(4) $\ulcorner A \urcorner \downarrow \mathsf{E}_{\mathrm{fo}}$

where A is a formula.

(5) $\ulcorner s \urcorner \downarrow \mathsf{E}_{\mathrm{sy}}$

where $s \in \mathcal{S}$.

(6) $\ulcorner o \urcorner \downarrow \mathsf{E}_{\mathrm{on}}$

where $o \in \mathcal{O}$.

Proof Parts $1 - 4$ follow from Axiom Schemas 13 in [9] and parts 5 and 6 are parts 1 and 2 of Axiom Schemas 11 in [9]. □

The following theorems deal with the definedness of evaluation.

Theorem 13 (Evaluation)

(1) $[\![\ulcorner \alpha \urcorner]\!]_{\mathrm{ty}} =_{\mathrm{ty}} \alpha$

where α is eval-free.

(2) $\neg \mathsf{gea}(b, \ulcorner \mathsf{type} \urcorner) \supset [\![b]\!]_{\mathrm{ty}} =_{\mathrm{ty}} \mathsf{C}$.

(3) $[\![\ulcorner a \urcorner]\!]_{\alpha} = \mathsf{if}(a \downarrow \alpha, a, \bot_{\mathsf{C}})$

where a is eval-free.

(4) $\neg \mathsf{gea}(b, \ulcorner \alpha \urcorner) \supset [\![b]\!]_{\alpha} \uparrow$.

(5) $[\![\ulcorner A \urcorner]\!]_{\mathrm{fo}} \equiv A$

where A is eval-free.

(6) $\neg \mathsf{gea}(b, \ulcorner \mathsf{formula} \urcorner) \supset [\![b]\!]_{\mathrm{fo}} \equiv \mathsf{F}$.

Proof This is Axiom Schemas 12 in [9]. □

CHAPTER 5

AND/OR TREES

5.1 Overview

From a broadest view, any computational task can be regarded as part of the larger problem process that includes problem solving. In artificial intelligence research, many problem-solving methods solve problems by searching for a solution in a space of possible solutions. One approach to problem solving is problem reduction. In this approach, we analyse the original problem and simplify it to some subproblems so that solutions to the subproblems produce a solution to the original problem.

The main purpose for us is to develop a Chiron definedness checking system for the MathScheme project that automatically reduces a definedness problem to simpler definedness problems that can be easily, or perhaps automatically, checked. Therefore, the definedness checking problem could be considered as a problem-reduction problem. As demonstrated in the literature in artificial intelligence [23], a good formalism for representing the problem-reduction approach to problem solving is based on AND/OR trees [19]. We can conveniently diagram the reduction of a definedness problem to alternative sets of problems by using three-valued AND/OR trees.

This chapter is divided into two main sections. The first section describes the main idea of AND/OR trees by a simple example. The second section presents how AND/OR trees can represent different categories of definedness problems in Chiron

and explains why those trees are sound based on the definedness theorems given in the Chapter 4.

5.2 Formalization of AND/OR Trees for Definedness Problems

An *AND/OR tree* is a representation of a reduction problem involving conjunctions and disjunctions of subproblems. Suppose, we have a problem A which can be solved either by solving problems B and C, or by solving problem D. Problems B and C constitute one set of problems, problem D another. The relationship is shown by the structure in Figure 5.1. There the nodes of this structure are labelled by the problems they represent.

In the figure, the node labelled E serves as the parent for problems B and C. Since the problem A is reduced to alternative subproblems D or E, the node labelled A is called an *OR node*, which is indicated with a single-lined square. Problem E, however, is reduced to a set of subproblems B and C, all of which must be solved to solve the parent node E. So the node labelled E is called an *AND node* and it indicated with a double-lined square. The nodes labelled B, C and D, which are indicated with an ellipse, are called *leaf nodes*. Structures like those shown in Figure 5.1 are called *AND/OR trees*.

We will often employ the following terms in describing AND/OR trees: *parent* nodes, *successor* nodes, *AND* nodes, *OR* nodes, and *leaf* nodes. The AND/OR trees use for a definedness checking system are the same as classical AND/OR trees except the individual nodes of the AND/OR trees hold Chiron formulas.

Definition 1 (AND node) *A node is an* AND node *if the problem it represents will be solved if all of the subproblems represented by its successors are solved.*

Definition 2 (OR node) *A node is an* OR node *if the problem it represents will be solved if any of the the subproblems represented by its successors are solved.*

Definition 3 (Leaf node) *A node is a* leaf node *if it does not have any successor.*

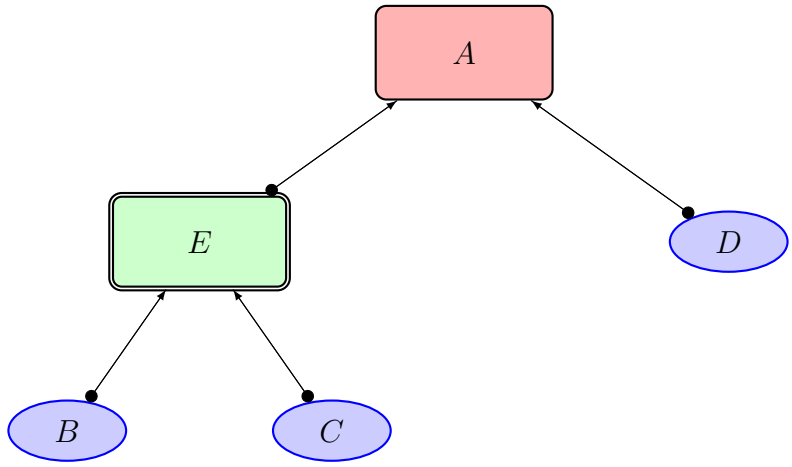

Figure 5.1: AND/OR tree for $D \vee (B \wedge C)$.

Definition 4 (Start node) *A* start node *is an OR node associated with a Chiron formula which represents the initial definedness problem. The start node is the root of an AND/OR tree.*

The nodes of classical AND/OR trees are usually marked as *solved* or *unsolved*. For definedness checking we mark AND/OR nodes differently. They are given values *true*, *false*, and *unknown* to represent the status of each node. The general definitions of the three values are as follows:

Definition 5 (True) *The status of a node is* true *if the formula held by the node is known to be true.*

Definition 6 (False) *The status of a node is* false *if the formula held by the node is known to be false.*

Definition 7 (Unknown) *The status of a node is* unknown *if the formula held by the node is not known.*

Now we present the propositions for checking the status of different nodes.

Proposition 1 (True)

- *The status of an AND node is true if all of the formulas held by its successors are known to be true.*

- *The status of an OR node is true if at least one of the formula held by its successors is known to be true.*

Proposition 2 (Unknown) *A node is an unknown node if it is neither true nor false.*

From these theorems, we can generate the node status truth tables which will be useful later for our definedness checking process. For an *AND node*, the truth table is as follows:

Table 5.1: AND node status truth table

AND	True	False	Unknown
True	True	Unknown	Unknown
False	Unknown	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown

And for an *OR node*, the truth table is as follows:

Table 5.2: OR node status truth table

OR	True	False	Unknown
True	True	True	True
False	True	Unknown	Unknown
Unknown	True	Unknown	Unknown

Then we need to prove that the AND/OR trees are *sound*. The definition of soundness is as follows:

Definition 8 (Sound) *An AND/OR tree is* sound *if whenever the leaf nodes are true the root of the tree is also true.*

A *solution tree* of an AND/OR Tree T is then defined to be any subtree S of T having the following properties:

- The root node of T is in S.

- Suppose a node A of T is an AND node and is included in S. Then all of the successors of A are also included in S.
- Suppose a node A of T is an OR node and is included in S. Then one and only one of the successors of A is also included in S.
- All of the leaf nodes in the solution graph have the status true.

Assume that in our example above the node B is an unknown node, and the nodes C and D are true nodes. Then the status of node E is unknown. Therefore, we show in Figure 5.2 a solution tree is indicated by the subtree whose edges are thick, i.e., in this case the tree composed of nodes A and D.

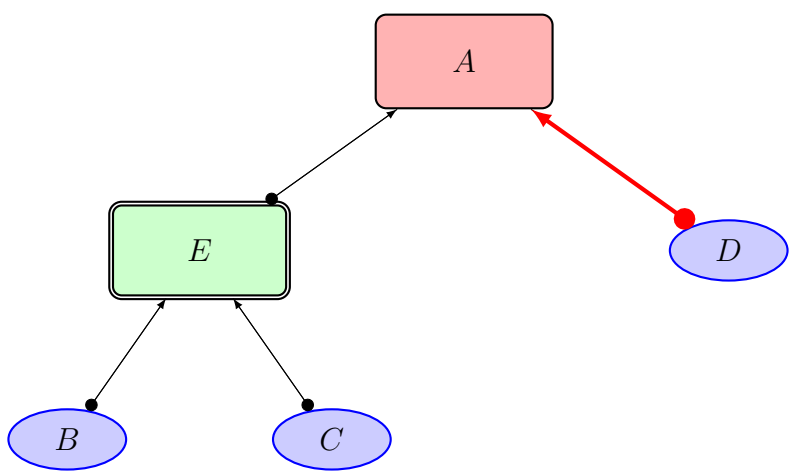

Figure 5.2: Example of a solution graph of an AND/OR tree.

5.3 Reducing Definedness Problems in Chiron Using Starter Trees

An AND/OR tree may be generated to reduce the definedness problems in Chiron. The generation of this tree comprises a *starter tree* building process that represent basic definedness problems reducing techniques.

Definition 9 (Starter tree) *An AND/OR tree is a* starter tree *if it represents the reduction of a definedness problem of basic Chiron proper expressions.*

In this section we shall present starter trees for reducing definedness problems in Chiron based on the proper expressions given in [9] and prove the soundness of those trees based on the theorems given in Chapter 4.

5.3.1 General Cases

Figure 5.3 is the AND/OR tree used to record how the assertion of a term a of type α defined in a type β is reduced.

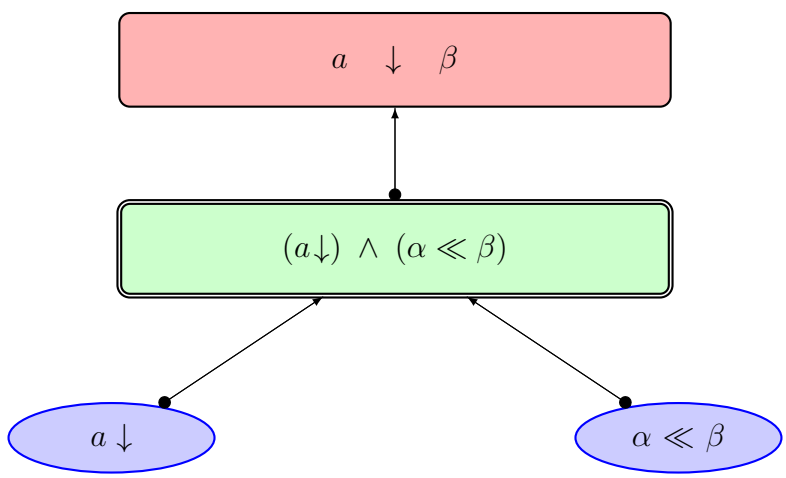

Figure 5.3: AND/OR tree for the general cases of being defined in a type.

Soundness Proof $a \downarrow \alpha$ follows immediately from $a\downarrow$ by Theorem 6(3). Then $a \downarrow \beta$ follows from $a \downarrow \alpha$ and $\alpha \ll \beta$ by Theorem 6(1). Therefore the formula of the root node is true when $a\downarrow$ and $\alpha \ll \beta$ are true, and so the tree in Figure 5.3 is sound. □

Figure 5.4 is the AND/OR tree used to record how the assertion of a term a is undefined in a type β is reduced.

Soundness Proof $a \uparrow \beta$ follows immediately from $a\uparrow$ by Theorem 6(5). Also, $a \uparrow \beta$ follows from $\beta =_{\text{ty}} \nabla$ by Theorem 6(2). Therefore the formula of the root node is true when $a\uparrow$ or $\beta =_{\text{ty}} \nabla$ is true, and so the tree in Figure 5.4 is sound. □

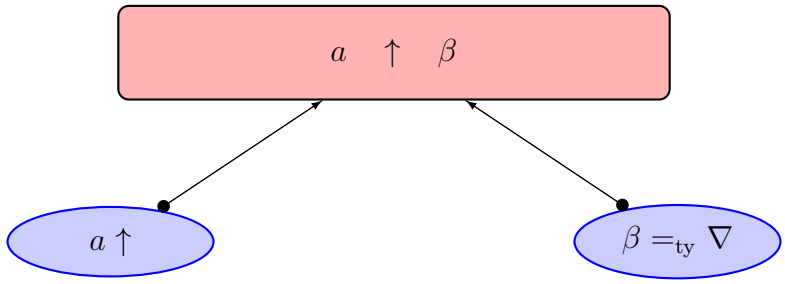

Figure 5.4: AND/OR tree for general cases undefined in a type.

5.3.2 Operator Application

If $o \in \mathcal{O}$ and $k_1, \ldots, k_{n+1}$ are kinds where $n \geq 0$, then

$$(\mathsf{op}, o, k_1, \ldots, k_{n+1})$$

is an n-ary *operator*.

Chiron has two kinds of operators, strict and non-strict. The definedness checker can check the context to see whether an operator is strict or non-strict, i.e., whether the context contains $\mathsf{strict}(o)$ or $\neg\mathsf{strict}(o)$.

Suppose $O = (\mathsf{op}, o, k_1, \ldots, k_n, k_{n+1})$ is an operator and $e_1 \ldots, e_n$ are expressions such that $k_i = \mathsf{type}$ and e_i is a type, k_i is a type and e_i is a term, or $k_i = \mathsf{formula}$ and e_i is a formula for all i with $1 \leq i \leq n$. Then

$$(\mathsf{op\text{-}app}, O, e_1, \ldots, e_n)$$

is an expression called an *operator application*. The operator application is a type if $k_{n+1} = \mathsf{type}$, a term of type k_{n+1} if k_{n+1} is a type, and a formula if $k_{n+1} = \mathsf{formula}$.

There are many different operators in Chiron, so it is important to consider the general facts about of operator applications. First, let us think about the type operator application. Figure 5.5 is the AND/OR tree used to record how the assertion of an operator application $(o :: k_1, \ldots, k_n, \mathsf{type})(e_1, \ldots, e_n)$ equals to C is reduced.[1]

[1]Notice that when we describe AND/OR trees, we use the Chiron compact notations for node contents whenever it is convenient.

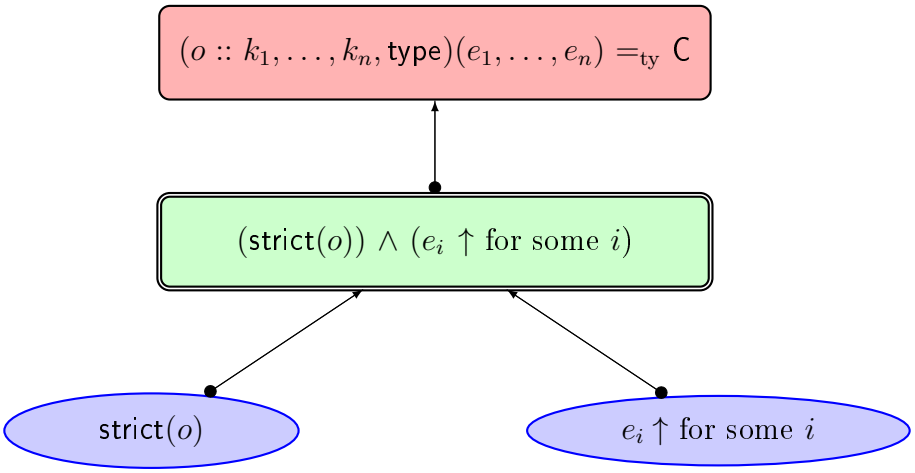

Figure 5.5: AND/OR tree for type operator applications equal to C.

Soundness Proof The soundness of the tree in Figure 5.5 follows from Theorem 3(5). □

Figure 5.6 is the AND/OR tree used to check whether a term operator application $(o :: k_1, \ldots, k_n, \beta)(e_1, \ldots, e_n)$ is undefined.

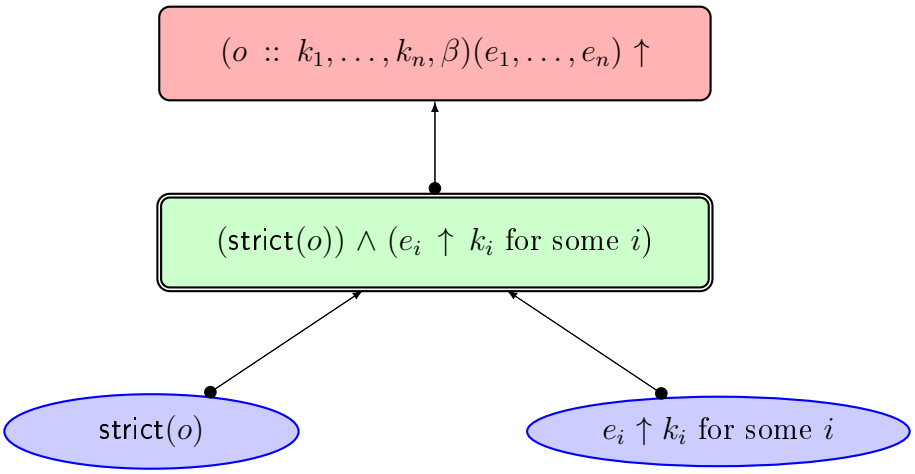

Figure 5.6: AND/OR tree for undefined term operator application.

Soundness Proof The soundness of the tree in Figure 5.6 follows from Theorem 3(7). □

Figure 5.7 is the AND/OR tree used to check whether a formula operator application $(o :: k_1, \ldots, k_n, \mathsf{formula})(e_1, \ldots, e_n)$ is false.

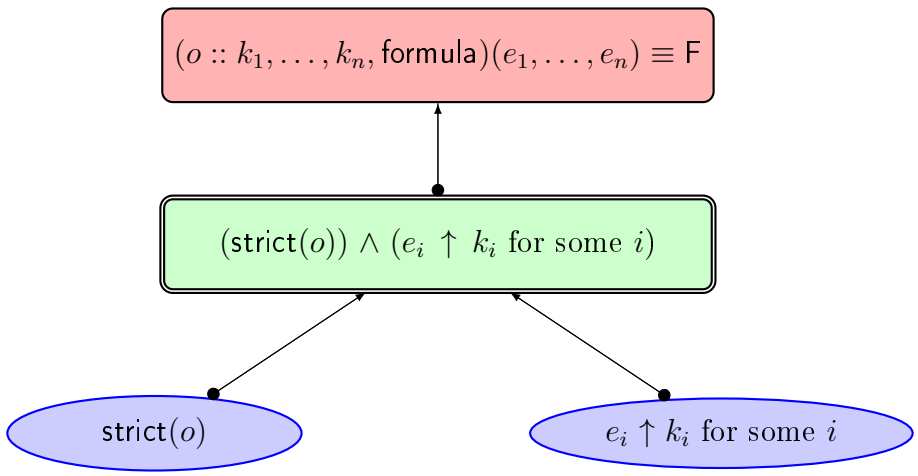

Figure 5.7: AND/OR tree for formula operator application false.

Soundness Proof The soundness of the tree in Figure5.7 follows from Theorem 3(9). □

5.3.3 Constants

Suppose $O = (\mathsf{op}, o, k)$ is an operator, then

$$(\mathsf{con}, o, k)$$

is a *constant* of type k. It is a type if k is type, a term if k is of type α, a formula if k is $\mathsf{formula}$.

Term constants are almost always defined, type constants are almost always nonempty. There are no applicable theorems, thus the definedness checker will check the context to see whether a term constant is defined or a type constant is nonempty.

5.3.4 Variables

If $x \in \mathcal{S}$ and α is a type, then

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

is a term of type α called a *variable*. As in traditional predicate logic, a variable may occur as either bound or free in a proper expression. The value of a bound variable $(\mathsf{var}, x, \alpha)$ is restricted to be in the type α. The value of a free variable $(\mathsf{var}, x, \alpha)$ is either in the type α or is the undefined value $\perp$.

The method to check the definedness problems of variables is covered under the general cases in Section 3. Figure 5.3 is the AND/OR tree used to check whether a variable is defined in a given type β. And Figure 5.4 is used to check whether a variable is undefined in a given type β.

5.3.5 Type Application

If α is a type and a is a term, then

$$(\mathsf{type\text{-}app}, \alpha, a)$$

is a type called a *type application*. If a is undefined, the type application denotes the domain D_{c} of all classes.

The general definedness AND/OR Trees for type applications are present as the following figures:

Figure 5.8 is the AND/OR tree used to record how the assertion of a type application equals to empty is reduced.

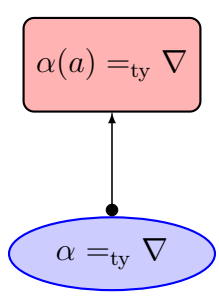

Figure 5.8: AND/OR tree for empty type applications.

Soundness Proof The soundness of the tree in Figure 5.8 follows from Theorem 7(1). □

Figure 5.9 is the AND/OR tree used to record how the assertion of a type application equals to C is reduced.

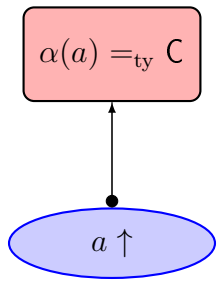

Figure 5.9: AND/OR tree for type applications equal to C.

Soundness Proof The soundness of the tree in Figure 5.9 follows from Theorem 7(2). □

5.3.6 Dependent Function Types

If $(\mathsf{var}, x, \alpha)$ is a variable and β is a type, then

$$(\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$$

is a type called a *dependent function type*.

The general definedness AND/OR trees for dependent function type are presented by the following figures.

Figure 5.10 is the AND/OR tree used to record how the assertion of a function defined in a dependent function type is reduced.

Soundness Proof The soundness of the tree in Figure 5.10 follows from Theorem 8(2). □

Figure 5.11 is the AND/OR tree used to record how the assertion of a function undefined in a dependent function type is reduced.

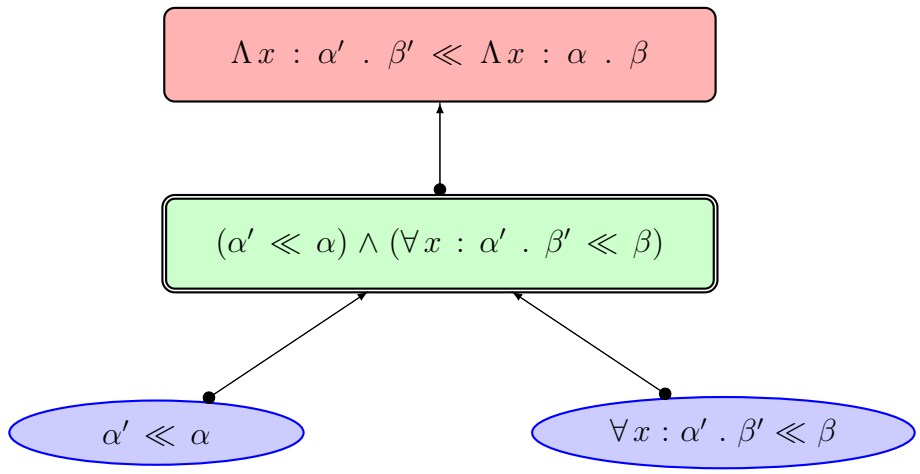

Figure 5.10: AND/OR tree for function is defined in a dependent function type.

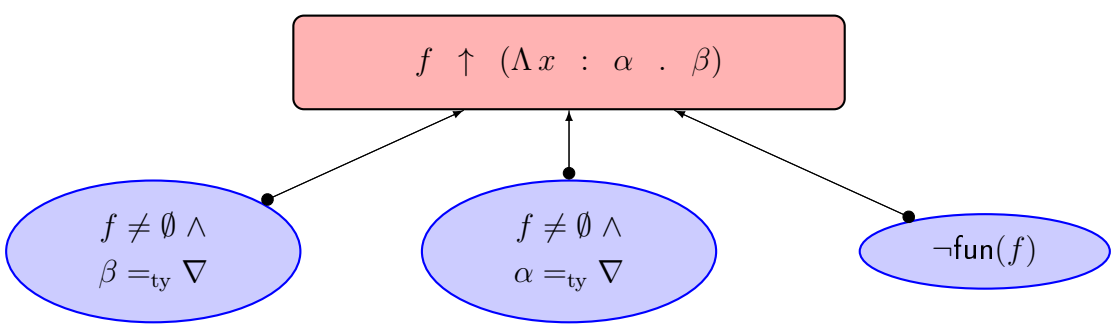

Figure 5.11: AND/OR tree for functions undefined in a dependent function type.

Soundness Proof

Based on the definition of a dependent function type, we know that if f is a term of type $\Lambda x : \alpha . \beta$ and a is a term of type α, then the application $f(a)$ is of type $(\Lambda x : \alpha . \beta)(a)$.

$f(a) \uparrow (\Lambda x : \alpha . \beta)(a)$ follows immediately from $f \neq \emptyset \wedge \beta =_{\text{ty}} \nabla$, and $f \uparrow (\Lambda x : \alpha . \beta)$ follows from $f(a) \uparrow (\Lambda x : \alpha . \beta)(a)$ by the definition of a dependent function type. Also, $a \uparrow \alpha$ follows immediately from $\mathsf{fun}(f) \neq \emptyset \wedge \alpha =_{\text{ty}} \nabla$ by Theorem 6(6), and $f \uparrow (\Lambda x : \alpha . \beta)$ follows from $a \uparrow \alpha$ by the definition of a dependent function type. And $f \uparrow (\Lambda x : \alpha . \beta)$ follows from $\neg\mathsf{fun}(f)$ by the same definition. Therefore

the formula of the root node is true when $\mathsf{fun}(f) \neq \emptyset \wedge \beta =_{\mathrm{ty}} \nabla$, $\mathsf{fun}(f) \neq \emptyset \wedge \alpha =_{\mathrm{ty}} \nabla$ or $\neg\mathsf{fun}(f)$ is true, and so the tree is sound. $\square$

5.3.7 Function Application

If f is a term of type γ and a is term of type α, then

$$(\mathsf{fun\text{-}app}, f, a)$$

is a term of type $(\mathsf{type\text{-}app}, \gamma, a)$ called a *function application*.

Figure 5.12 is the AND/OR tree used to record how the assertion of a function application defined in a given type is reduced.

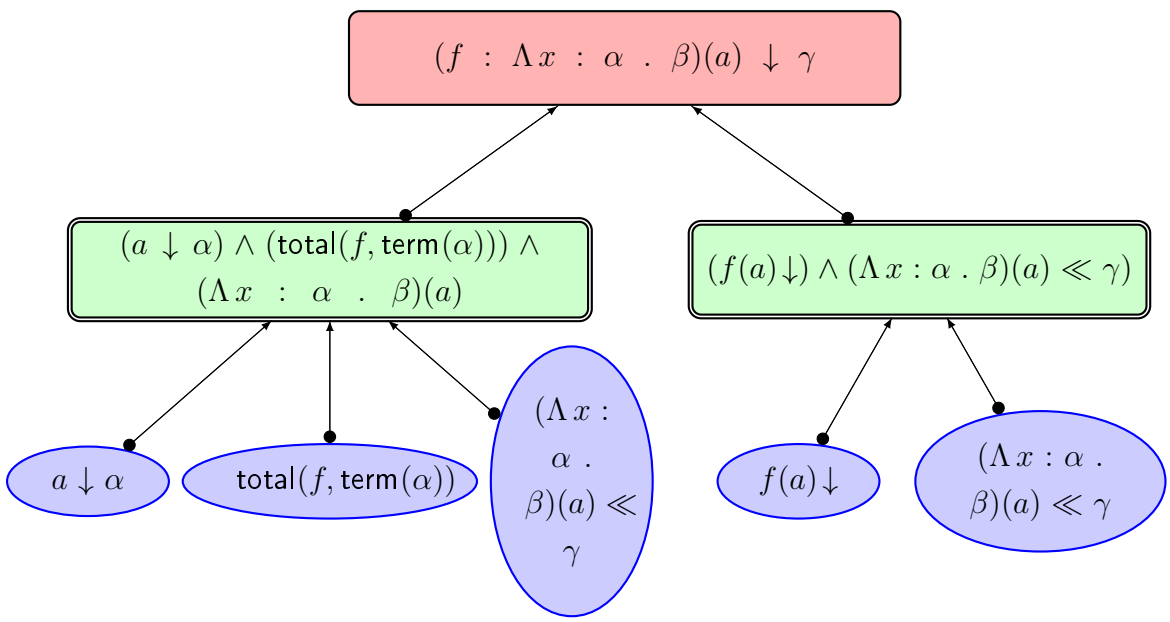

Figure 5.12: AND/OR tree for function application is defined in a given type.

Soundness Proof

Part 1 In mathematics, a function f from α to β is total if the domain of f is the full denotation of α. $(f : \Lambda x : \alpha . \beta)(a) \downarrow (\Lambda x : \alpha . \beta)(a)$ follows immediately from $a \downarrow \alpha$ and $\mathsf{total}(f, \mathsf{term}(\alpha))$ by Theorem 6(1) and the definition of a total function. Then $(f : \Lambda x : \alpha . \beta)(a) \downarrow \gamma$ follows from $(\Lambda x : \alpha . \beta)(a) \ll \gamma$

and $(f : \Lambda\, x : \alpha\ .\ \beta)(a) \downarrow (\Lambda\, x : \alpha\ .\ \beta)(a)$ by Theorem 6(1). Therefore the formula of the root node is true when $a \downarrow \alpha$, $\mathsf{total}(f, \mathsf{term}(\alpha))$ and $(\Lambda\, x : \alpha\ .\ \beta)(a) \ll \gamma$ are true.

Part 2 $(f : \Lambda\, x : \alpha\ .\ \beta)(a) \downarrow (\Lambda\, x : \alpha\ .\ \beta)(a)$ follows immediately from $f(a)\downarrow$ by the Theorem 6(3). Then $(f : \Lambda\, x : \alpha\ .\ \beta)(a) \downarrow \gamma$ follows from $(\Lambda\, x : \alpha\ .\ \beta)(a) \ll \gamma$ and $(f : \Lambda\, x : \alpha\ .\ \beta)(a) \downarrow (\Lambda\, x : \alpha\ .\ \beta)(a)$ by the Theorem 6(1). Therefore the formula of the root node is true when $f(a)\downarrow$ and $(\Lambda\, x : \alpha\ .\ \beta)(a) \ll \gamma$ are true.

So the formula of the root node is true when the part 1 or the part 2 is true, and so the tree in Figure 5.12 is sound. □

Figure 5.13 is the AND/OR tree used to record how the assertion of a function application undefined in a given type is reduced.

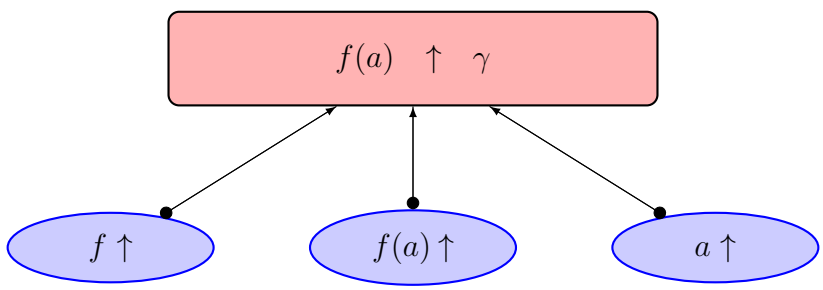

Figure 5.13: AND/OR tree for function application is undefined in a given type.

Soundness Proof $f(a)\uparrow$ follows immediately from $f\uparrow$ or $a\uparrow$ by the definition of a total function, and $f(a) \uparrow \gamma$ follows from $f(a)\uparrow$ by Theorem 6(5). Therefore the formula of the root node is true when $f\uparrow$, $f(a)\uparrow$, or $a\uparrow$ is true, and so the tree in Figure 5.13 is sound. □

5.3.8 Function Abstraction

If $(\mathsf{var}, x, \alpha)$ is a variable and b is a term of type β, then

$$(\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$$

is a term of type $(\mathsf{dep\text{-}fun\ type}, (\mathsf{var}, x, \alpha), \beta)$ called a *function abstraction*.

Figure 5.14 is the AND/OR tree used to record how the assertion of a function abstraction is defined is reduced.

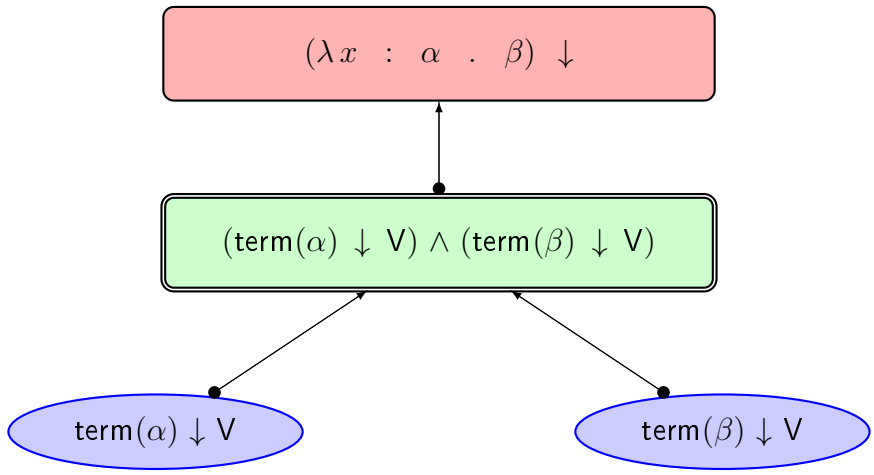

Figure 5.14: AND/OR tree for function abstraction is defined.

Soundness Proof The soundness of the tree in Figure 5.14 follows from the semantics of a function abstraction. □

5.3.9 Conditional Terms

If A is a formula and b, c are terms, then

$$(\mathsf{if}, A, b, c)$$

is an if-then-else term called a *conditional term*.

Figure 5.15 is the AND/OR tree used to record how the assertion of a conditional term defined in a given Type is reduced.

Soundness Proof

Part 1 $\mathsf{if}(A, b, c) \simeq b$ follows immediately from A by Theorem 9(1). Then $\mathsf{if}(A, b, c) \downarrow \beta$ follows from $\mathsf{if}(A, b, c) \simeq b$ and $b \downarrow \beta$ by Theorem 1(2). Next, $(\mathsf{if}, A, b, c) \downarrow \alpha$ follows from $\mathsf{if}(A, b, c) \downarrow \beta$ and $\beta \ll \alpha$ by Theorem 6(1). Therefore the formula of

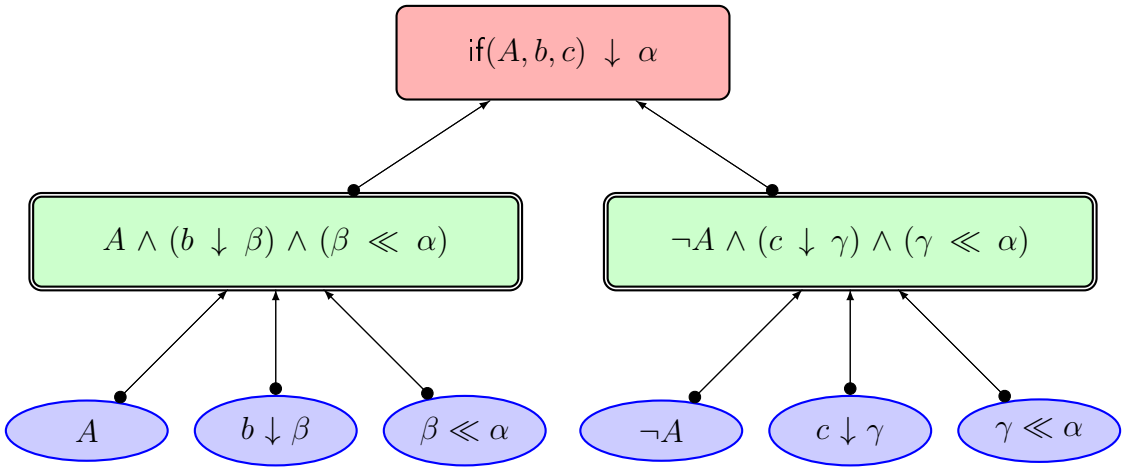

Figure 5.15: AND/OR tree for conditional term is defined in a given type.

the root node is true when A, $b \downarrow \beta$ and $\beta \ll \alpha$ are true.

Part 2 $\mathsf{if}(A, b, c) \simeq c$ follows immediately from $\neg A$ by Theorem 9(2). Then $\mathsf{if}(A, b, c) \downarrow \beta$ follows from $\mathsf{if}(A, b, c) \simeq c$ and $c \downarrow \gamma$ by Theorem 1(2). Next, $\mathsf{if}(A, b, c) \downarrow \alpha$ follows from $\mathsf{if}(A, b, c) \downarrow \beta$ and $\gamma \ll \alpha$ by Theorem 6(1). Therefore the formula of the root node is true when $\neg A$, $c \downarrow \gamma$ and $\gamma \ll \alpha$ are true.

So the formula of the root node is true when the part 1 or the part 2 is true, and so the tree in Figure 5.15 is sound. $\square$

Figure 5.16 is the AND/OR tree used to record how the assertion of a conditional term undefined in a given type is reduced.

Soundness Proof

Part 1 $\mathsf{if}(A, b, c) \uparrow \alpha$ follows from $\alpha =_{\mathrm{ty}} \nabla$ by Theorem 5(2).

Part 2 $\mathsf{if}(A, b, c) \simeq b$ follows immediately from A by Theorem 9(1). Then $\mathsf{if}(A, b, c) \uparrow$ follows from $\mathsf{if}(A, b, c) \simeq b$ and $b \uparrow$ by Theorem 1(2). Next, $\mathsf{if}(A, b, c) \uparrow \alpha$ follows from $\mathsf{if}(A, b, c) \uparrow$ by Theorem 6(5). Therefore the formula of the root node is true when A and $b \uparrow$ are true.

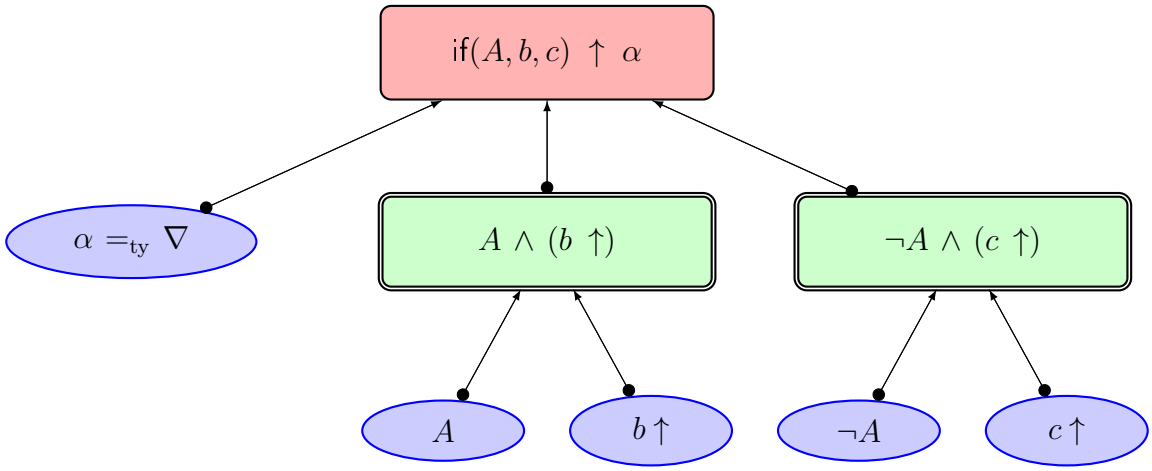

Figure 5.16: AND/OR tree for conditional term is undefined in a given type.

Part 3 $\mathsf{if}(A, b, c) \simeq c$ follows immediately from D by Theorem 9(2). Then $\mathsf{if}(A, b, c)\uparrow$ follows from $\mathsf{if}(A, b, c) \simeq c$ and $c\uparrow$ by Theorem 1(2). Next, $\mathsf{if}(A, b, c) \uparrow \alpha$ follows from $\mathsf{if}(A, b, c)\uparrow$ by Theorem 6(5). Therefore the formula of the root node is true when D and $c\uparrow$ are true.

So the formula of the root node is true when one case of the part 1, part 2, or part 3 is true, and so the tree in Figure 5.16 is sound. □

5.3.10 Definite Description

If $(\mathsf{var}, x, \alpha)$ is a variable and B is a formula, then

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

is a term called a *definite description*.

Figure 5.17 is the AND/OR tree used to record how the assertion of a definite description defined in a given Type is reduced.

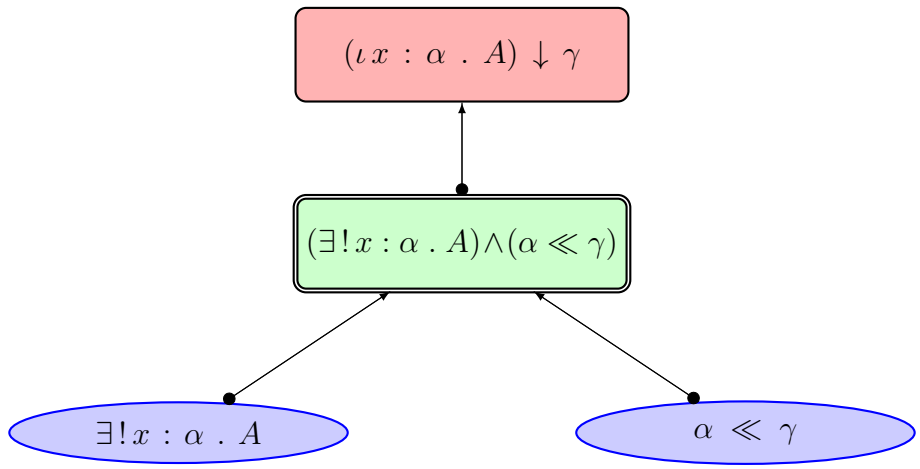

Figure 5.17: AND/OR tree for definite description is defined in a given type.

Soundness Proof $(\iota x : \alpha \,.\, A) \downarrow \alpha$ follows immediately from $\exists! x : \alpha \,.\, A$ by Theorem 10(1). Then $(\iota x : \alpha \,.\, A) \downarrow \gamma$ follows from $(\iota x : \alpha \,.\, A) \downarrow \alpha$ and $\alpha \ll \gamma$ by Theorem 6(1). Therefore the formula of the root node is true when $\exists! x : \alpha \,.\, A$ and $\alpha \ll \gamma$ are true, and so the tree in Figure 5.17 is sound. □

Figure 5.18 is the AND/OR tree used to record how the assertion of a definite description undefined in a given Type is reduced.

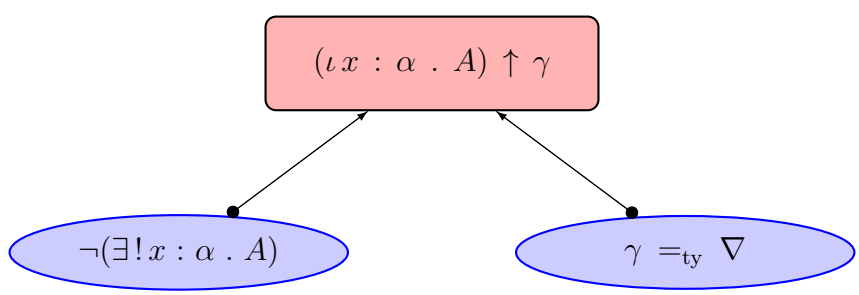

Figure 5.18: AND/OR tree for definite description is undefined in a given type.

Soundness Proof $(\iota x : \alpha \,.\, A) \uparrow$ follows immediately from $\neg(\exists! x : \alpha \,.\, A)$ by Theorem 10(2), and $(\iota x : \alpha \,.\, A) \uparrow \gamma$ follows from $(\iota x : \alpha \,.\, A) \uparrow$ by Theorem 6(5). Also, $(\iota x : \alpha \,.\, A) \uparrow \gamma$ follows from $\gamma =_{\mathrm{ty}} \nabla$ by Theorem 5(2). Therefore the formula of the root node is true when $\neg(\exists! x : \alpha \,.\, A)$ or $\gamma =_{\mathrm{ty}} \nabla$ is true, and so the tree in Figure 5.18 is sound. □

5.3.11 Indefinite Description

If $(\mathsf{var}, x, \alpha)$ is a variable and B is a formula, then

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

is a term called an *indefinite description*.

Figure 5.19 is the AND/OR tree used to record how the assertion of a indefinite description defined in a given Type is reduced.

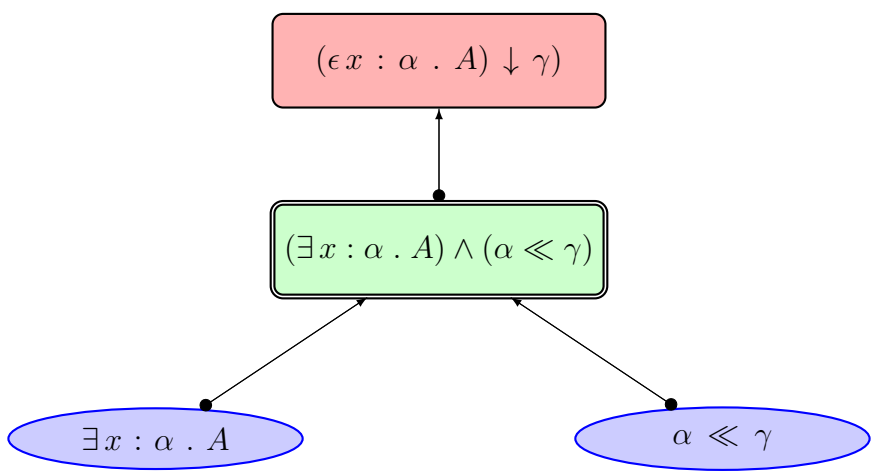

Figure 5.19: AND/OR tree for indefinite description is defined in a given type.

Soundness Proof $(\epsilon\, x : \alpha \;.\; A) \downarrow \alpha$ follows immediately from $\exists\, x : \alpha \;.\; A$ by Theorem 11(1). Then $(\epsilon\, x : \alpha \;.\; A) \downarrow \gamma$ follows from $(\epsilon\, x : \alpha \;.\; A) \downarrow \alpha$ and $\alpha \ll \gamma$ by Theorem 6(1). Therefore the formula of the root node is true when $\exists\, x : \alpha \;.\; A$ and $\alpha \ll \gamma$ are true, and so the tree in Figure 5.19 is sound. $\square$

Figure 5.20 is the AND/OR tree used to record how the assertion of a indefinite description undefined in a given Type is reduced.

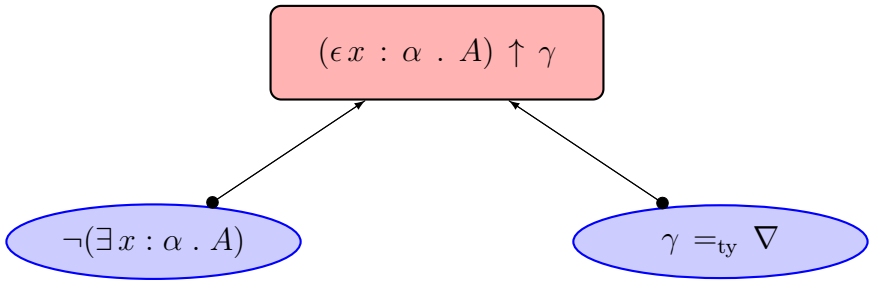

Figure 5.20: AND/OR tree for indefinite description is undefined in a given type.

Soundness Proof $(\epsilon\, x : \alpha \,.\, A) \uparrow$ follows immediately from $\neg(\exists\, x : \alpha \,.\, A)$ by Theorem 11(2), and $(\epsilon\, x : \alpha \,.\, A) \uparrow \gamma$ follows from $(\epsilon\, x : \alpha \,.\, A) \uparrow$ by Theorem 6(5). Also, $(\epsilon\, x : \alpha \,.\, A) \uparrow \gamma$ follows from $\gamma =_{\mathrm{ty}} \nabla$ by Theorem 5(2). Therefore the formula of the root node is true when $\neg(\exists\, x : \alpha \,.\, A)$ or $\gamma =_{\mathrm{ty}} \nabla$ is true, and so the tree in Figure 5.20 is sound. □

5.3.12 Quotation

If e is any expression, proper or improper, then

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

is a term of expression constructions type

$$(\mathsf{op\text{-}app}, (\mathsf{op}, \mathsf{expr}, \mathsf{type}))$$

called a *quotation*.

Every quotation is defined. Figure 5.21 is the AND/OR tree used to record how the assertion of a type quotation defined in a type β is reduced.

Soundness Proof By the theorem 12(1), $\ulcorner\alpha\urcorner \downarrow \mathsf{E}_{\mathrm{ty}}$ is true. $\ulcorner\alpha\urcorner \downarrow \beta$ follows immediately from $\ulcorner\alpha\urcorner \downarrow \mathsf{E}_{\mathrm{ty}}$ and $\mathsf{E}_{\mathrm{ty}} \ll \beta$ by Theorem 6(1). Therefore the formula of the root node is true when $\mathsf{E}_{\mathrm{ty}} \ll \beta$ is true, and so the tree in Figure 5.21 is sound. □

Figure 5.22 is the AND/OR tree used to record how the assertion of a term quotation of type α defined in a type β is reduced.

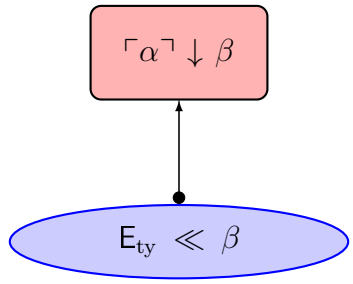

Figure 5.21: AND/OR tree for type quotation is defined in a given type.

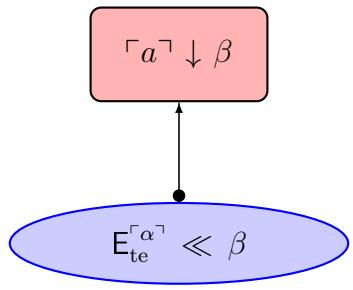

Figure 5.22: AND/OR tree for term quotation is defined in a given type.

Soundness Proof By the Theorem 12(3), $\ulcorner a\urcorner \downarrow \mathsf{E}_{\mathrm{te}}^{\ulcorner\alpha\urcorner}$ is true. $\ulcorner a\urcorner \downarrow \beta$ follows immediately from $\ulcorner a\urcorner \downarrow \mathsf{E}_{\mathrm{te}}^{\ulcorner\alpha\urcorner}$ and $\mathsf{E}_{\mathrm{te}}^{\ulcorner\alpha\urcorner} \ll \beta$ by Theorem 6(1). Therefore the formula of the root node is true when $\mathsf{E}_{\mathrm{te}}^{\ulcorner\alpha\urcorner} \ll \beta$ is true, and so the tree in Figure 5.22 is sound. □

Other categories of quotations are handled by the same way as above.

5.3.13 Evaluation

If a is a term and k is a kind, then

$$(\mathsf{eval}, a, k)$$

is an expression called an *evaluation* that is a type if $k = \mathsf{type}$, a term of type k if k is a type, and a formula if $k = \mathsf{formula}$.

Figure 5.23 is the AND/OR tree used to record the assertion of a type evaluation equals to C is reduced.

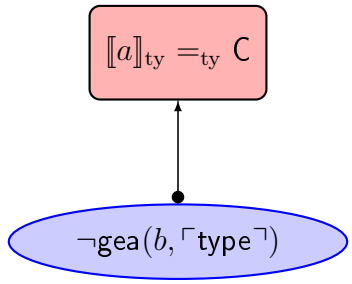

Figure 5.23: AND/OR tree for type evaluation is of C type.

Soundness Proof The soundness of the tree in Figure 5.23 follows from Theorem 13(2). □

Figure 5.24 is the AND/OR tree used to record how the assertion of a term evaluation is undefined is reduced.

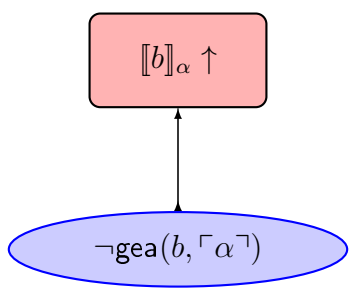

Figure 5.24: AND/OR tree for term evaluation is undefined.

Soundness Proof The soundness of the tree in Figure 5.24 follows from Theorem 13(4). □

Figure 5.25 is the AND/OR tree used to record how the assertion of a formula evaluation is false is reduced.

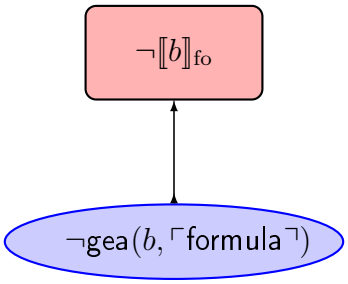

Figure 5.25: AND/OR tree for formula evaluation is false.

Soundness Proof The soundness of the tree in Figure 5.25 follows from Theorem 13(6). □

CHAPTER 6

DEFINEDNESS CHECKING MECHANISM

6.1 Overview

Since the AND/OR trees may be very large, one job of the definedness checking mechanism is to avoid the full development of the AND/OR tree implied by the problem. Therefore, as the tree grows, decisions must be made about which leaf nodes to reduce.

In this chapter, we revisit different search methods such as breath-first, depth-first, and best-first methods discussed in Mahanti and Bagchi's work [15] and Nillson's book [19]. We use similar ideas to present a novel algorithm for our definedness checking system.

Section 2 discusses some different search methods, breadth-first, depth-first search, and ordered-search methods for AND/OR trees. Section 3 presents a new algorithm for checking the definedness of Chiron proper expressions.

6.2 Classical AND/OR Tree Search Methods

In this section, we shall be concerned with various AND/OR tree searching processes that efficiently order the expansion of nodes and do termination checks. The breadth-first methods expand nodes in the order of their successors, whereas the depth-first search methods expand the most recently generated nodes first. To start this section, we first briefly define the terminology of classical AND/OR trees used in this chapter. In the later subsections, we will show the sequence of steps for different search methods.

6.2.1 Definitions and Notation

For definedness checking we give values *true*, *false*, and *unknown*, which are defined in Chapter 5 to represent the status of AND/OR nodes. But the nodes of classical AND/OR trees are usually marked as *solved* or *unsolved*. The general definitions of classical AND/OR trees and its algorithms are as follows:

Definition 10 (Start node) *A* start (or root) node *is a node that is associated with the original problem description and is denoted by* s.

Definition 11 (Terminal node) *A* terminal node *is a leaf node that corresponds to a primitive problem or a problem with a known solution.*

Definition 12 (Nonterminal node) *A* nonterminal node *is a leaf node that is not a terminal node.*

Definition 13 (Solved node)

- *A terminal node is a* solved node.
- *A OR node is a* solved node *if at least one of its successors is solved.*
- *An AND node is a* solved node *if all of its successors are solved.*

Definition 14 (Unsolved node)

- *A nonterminal node is an* unsolved node.
- *An OR node is an* unsolved node *if all of its successors are* unsolved nodes.

- *An AND node is an* unsolved node *if at least one of its successors is an* unsolved node.

Definition 15 (OPEN) OPEN *is a finite set of nodes ready for expansion.*

Definition 16 (CLOSED) CLOSED *is a finite set of nodes that already have had the expansion procedure applied to them.*

Definition 17 (Labeling procedure) *A* labeling procedure *is used in simple recursive or iterative procedures that operate on an AND/OR tree to label all of the solved nodes.*

Definition 18 (Depth)

- *The* depth *of the start node is* 0.
- *The* depth *of any other node is* 1 *plus the depth of its parent.*

As mentioned in Chapter 5, for any node m in an implicit AND/OR tree T, a solution tree $D(m)$ with root m is a finite subtree of T defined as:

(1) m is in $D(m)$.

(2) If n is an OR node in T and n is in $D(m)$, then exactly one of its successors in T is in $D(m)$.

(3) If n is an AND node in T and n is in $D(m)$, then all its immediate successors in T are in $D(m)$.

(4) No other nodes in T are in $D(m)$.

6.2.2 Breadth-First Search

The breadth-first method expands nodes in the order in which they have been generated. Most of the steps are concerned with the termination checks. A simple algorithm for searching trees by breadth-first method consists of the following sequence of steps:

(1) Put the start node s on a list called OPEN.

(2) If OPEN is empty, exit with failure; otherwise continue.

(3) Remove the first node on OPEN and put it on a list called CLOSED; call this node n.

(4) Expand node n, generating all of its successors.

   (a) Put the successors at the end of OPEN and provide pointers from these successors back to n.

   (b) If there are no successors, label n unsolved and continue; otherwise go to (7).

(5) Apply the labeling procedure to the search tree.

   (a) If the start node is labeled unsolved, exit with failure; otherwise continue.

   (b) If the start node is labeled solved, exit with the solution tree that verifies that the start node is solved; otherwise continue.

   (c) Remove from OPEN any solved and unsolved nodes.

(6) Go to (3).

(7) If any of the successors are solved nodes, label them solved and go to (5); otherwise go to (3).

An example of how breadth-first procedure expands nodes is illustrated in Figure 6.1. The numbers held by the nodes indicate the order of node expansions and the solution tree found is indicated by the thick branches.

6.2.3 Depth-First Search

A depth-first search method expands the most recently generated nodes first. It attempts to find a solution tree within a certain depth bound, hence there is no node deeper than the depth bound is expanded. A simple algorithm for searching trees depth-first consists of the following sequence of steps:

(1) Put the start node s on a list called OPEN.

(2) If OPEN is empty, exit with failure; otherwise continue.

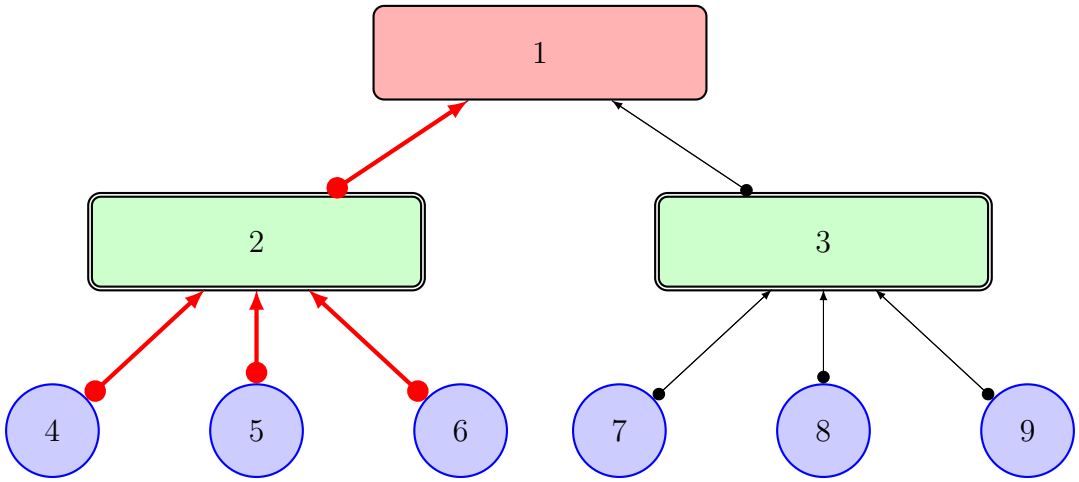

Figure 6.1: An AND/OR tree showing the order of node expansions in a breadth-first search.

(3) Remove the first node on OPEN and put it on a list called CLOSED; call this node n.

(4) If the depth of n is equal to the depth bound, label n unsolved and go to (5); otherwise continue.

(5) Expand node n, generating all of its successors.

   (a) Put these successors (in arbitrary order) at the beginning of OPEN and provide pointers back to n.

   (b) If there are no successors, label n unsolved and continue; otherwise go to (8).

(6) Apply the labeling procedure to the search tree.

   (a) If the start node is labeled unsolved, exit with failure; otherwise continue.

   (b) If the start node is labeled solved, exit with the solution tree that verifies that the start node is solved; otherwise continue.

   (c) Remove from OPEN any solved and unsolved nodes.

(7) Go to (3).

(8) If any of the successors are solved nodes, label them solved and go to (6); otherwise go to (3).

An example of how depth-first procedure expands nodes is illustrated in Figure 6.2. In this example the depth bound is set at 2, the numbers held by the nodes indicate the order of node expansions and the solution tree found is indicated by the thick branches.

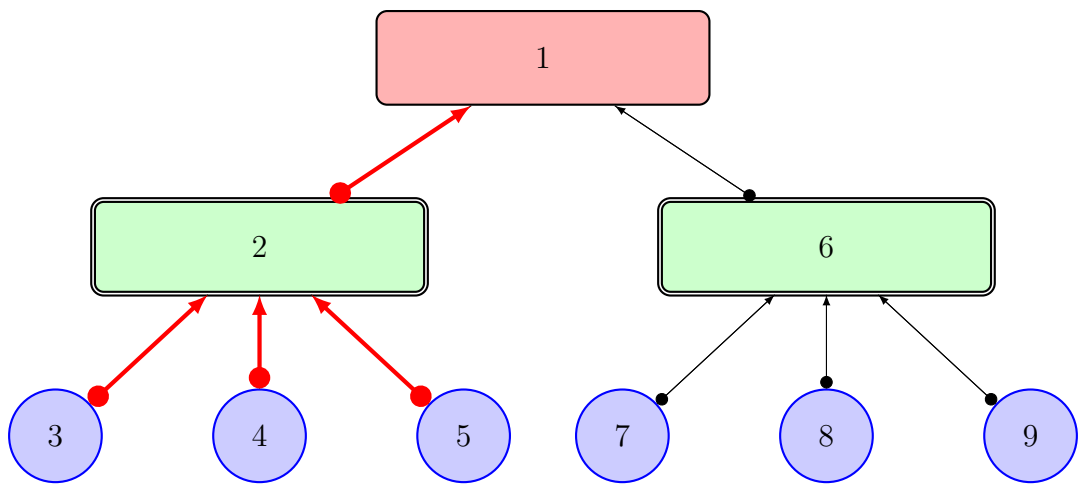

Figure 6.2: An AND/OR tree showing the order of node expansions in a depth-first search (depth-bound = 2).

6.2.4 Other Search Methods

The ordered-search method is another algorithm for AND/OR trees, which is used in searching for solution trees achieving minimal cost (either sum cost or max cost, depending on the situation). In this algorithm we shall make use of the idea of the estimated cost and heuristic function of a solution tree.

6.3 Definedness Checking Algorithm

An AND/OR tree search problem is to find a path from an initial state to a final state, and the solution of the search problem is a path from the initial state to the final state. The solution always has the backtracking process. This problem can be solved in two ways:

- Forward search: The search starts the search from the initial state and a path is built stepwise in the forward direction towards the final state.
- Backward search: The search starts the search from the final state and a path is built in the backward direction towards the initial state.

Either breadth-first, depth-first, or ordered-search methods can be used in Chiron definedness checking system. Depth-first search is an aggressive but dangerous procedure when a tree is complicated. When a tree has many levels, depth-first search, while easily implemented, it can waste resources to an incredible degree. When depth-first search is not a good idea, breadth-first search can be considered. The breadth-first search is wasteful when all paths lead to the destination node at more or less the same depth, but it is a careful and conservative method. The ordered-search is useful in searching for solution tree achieving minimal cost.

The objective for us is to generate an AND/OR tree for the definedness checking system that indicates the reduction of a definedness problem to alternative sets of problems that can be easily, or perhaps automatically, checked. Since the starter trees are not big, considering the time-limit, the storage-space limit, and the depth of the deepest node in the search tree, we combine the careful and conservative breadth-first methods with backward search to check the definedness problems.

6.3.1 Overview of Definedness Checking Algorithm

The resulting algorithm is decomposed in four phases:

(1) First, it builds a starter tree for a particular definedness checking problem.

(2) Then it applies the leaf checking procedure, using the breadth-first method, to check whether a primitive problem is solved. There are two ways to check the leaf nodes:

- Check if the context implies that the node's formula is true or false.
- Check the simplifier to see whether the node's formula is reduced to a new formula.

(3) Next, it applies the node labeling process based on the propositions of AND/OR trees to update ancestor nodes.

(4) In case the primitive problem cannot be solved by the tree, it finally starts expanding the leaf nodes of the tree.

6.3.2 Steps of Definedness Checking Algorithm

The structure of the procedure is quite simple as can be seen as the algorithm consists of the following steps:

(1) Match the input formula with the start nodes of starter trees, then build the corresponding tree.

(2) Execute leaf checking procedure.

  (a) Put the start node (or root) s on queue called q.

  (b) If q is empty, exit with failure; otherwise continue.

  (c) Remove the first node on q; call this node n.

  (d) Expand node n, generating all of its successors. Put the successors in q. If there are no successors, exit with unknown; If n is a leaf node, continue; otherwise go to (2)(c).

  (e) Check if the formula held by n is already stored in the context. If it is in the context, label it *true* and go to (4); if n is not in the context, continue.

  (f) Call the simplifier to see if the formula held by n reduces to a truth value or a simpler formula. If the formula held by n is simplified to true, label n true and go to (4); if the formula held by n is simplified to false, label n false and go to (4); if the formula held by n is reduces to a simpler formula which is not a truth value, replace the formula held by n with the new formula and go to (2)(e); otherwise, continue.

(3) Perform the procedure to expand an AND/OR tree.

  (a) Match the formula of n with the start nodes of starter trees.

  (b) If there is such a starter tree, expand n by building the corresponding tree and go to (2); otherwise, label n *unknown* and continue.

(4) Execute the node labeling procedure.

   (a) Apply the node labeling procedure to the tree.

   (b) If the start node is labeled true, exit with true; if the start node is labeled false, exit with false; if the start node is labeled unknown, exit with unknown; otherwise continue.

(5) Remove from q any nodes that are true, false, or unknown. (This step allows us to avoid the unnecessary effort that would be expended in attempting to solve unsolved problems.)

(6) Go to (2).

CHAPTER 7

IMPLEMENTATION

7.1 Overview

We shall repeat here the four phases of the definedness checking algorithm that were given in the last chapter.

(1) First, it builds a starter tree for a particular definedness checking problem.

(2) Then it applies the leaf checking procedure, using the breadth-first method, to check whether the primitive problem is solved. There are two ways to check the leaf nodes:

- Checking if the the node's formula is already stored in the context.
- Calling the simplifier to see whether the node's formula is reduced to a new formula.

(3) Next it applies the node labeling process to update ancestor nodes;

(4) In case the primitive problem can not be solved by the tree, it will expand the leaf nodes of the tree.

In this chapter, we describe the implementation of the application of AND/OR tree search algorithm to the definedness problem. Since the Chiron definedness checking system works on the Objective Caml (OCaml) implementation of Chiron formula expression, it is thus natural to implement our system in OCaml.

7.2 Implementation

7.2.1 Building Starter Trees

Phase 1 of definedness checking algorithm is implemented as a function which turns a formula into a starter tree of which each node holds a formula.

We first introduce the type of nodes in an AND/OR trees.

```
type status = Unchecked | True | False | Unknown

type node_rec = {mutable content : 'p formula; mutable status : status}

type node =
  | Leaf of node_rec * node option
  | And of node_rec * node option * (node list) ref
  | Or of node_rec * node option * (node list) ref
```

Note that `node_rec` is a record type of nodes which contains the formulas of nodes and the status (unchecked, true, false, or unknown) of nodes. Here, we use a new status `Unchecked` for representing the initial status of a node. The content and status are marked as `mutable`, in which case their value can be changed.

The first argument of each constant of type `node` is a `node_rec`; the second argument indicates the parent of the node; and the last argument is a list containing the children of the node. Here, we use a reference type to implement the list of children so that the list can be easily updated after a tree is expanded.

We implement phase 1 with the function `get_root`. It matches an input formula with the roots of starter trees and builds a starter tree based on the trees given in Chapter 5. If an input formula does not match any starter tree, it can not be checked.

```
let rec get_root checkExpr =
  match checkExpr with
  |...
```

```
...
| _ -> failwith "Can not be checked."
```

7.2.2 Leaf Checking Procedure

We implement phase 2 of the algorithm as two steps, first it uses a bread-first method to search the starter tree built in phase 1 by the function `get_root` until a leaf node is met, then it starts leaf checking by two methods: match the formula with the context or simplify the formula.

Search for the leaf nodes

Function `expand_root` builds a queue q, puts the root of the AND/OR tree in the queue, and then uses breadth-first method to search the AND/OR tree.

```
let rec expand_root n c =
  let q = Queue.create() in
    Queue.add (n) q;
    let rec bfs ns = ... in
    let nr = get_rec n in
      getStatus nr;;
```

In the above function, if the node taken from the queue is matched to an OR node, we scan the list of children and add each node to the queue.

```
    let rec bfs ns =
      let nr = get_rec n in
      let ns = getStatus nr in
        while ns = Unchecked do
          if Queue.length q = 0 then raise Not_found;
          let s = Queue.take q in
            match s with
              | Or (_, par, ch) ->
```

```
            for i = 0 to (List.length !ch) - 1 do
              Queue.add (List.nth !ch i) q
            done;
            bfs ns
```

If it is matched to an AND node, we return the following corresponding result.

```
            | And (_, par, ch) ->
              for i = 0 to (List.length !ch) - 1 do
                Queue.add (List.nth !ch i) q
              done;
              bfs ns
```

Otherwise, we start the second step to check the Leaf node.

```
            | Leaf (r, par) -> check_leaf r par c
      done
  and check_leaf r par c =
       ...
        bfs (updateNodes (get_parent par))
```

Check Leaf nodes

First, we match the node with the context to see whether the node is true or false. Thus, the beginning of the function is as follows.

```
  check_leaf r par c =
      while r.status = Unknown do
        let lc1 = getContent r in
          if inContext lc1 c then
            r.status <- True
```

```
        else
        ...
```

Then, if the node is not in the context, the simplifier will reduce the node to a new formula.

```
  let lc2 = simp_formula lc1 in
    r.content <- lc2;
               if (lc2 = R.falsef) then r.status <- False
                   else if (lc2 = R.truef) || (inContext lc2 c)
                            then r.status <- True
else
...
```

Note that, the above function checks the context again to update the status of nodes.

The leaf checking phase always needs to update the trees to see whether the root is true or false. Thus we need phase 3 to update the nodes of the trees.

7.2.3 Node Labeling Procedure

We now discuss the implementation of phase 3, which gets the status of a node and updates the status of the parents and the ancestors bottom-up.

```
let rec updateNodes n =
  match n with
  ...
```

In the above function, if `n` is matched to an OR node, we return the corresponding result.

```
| Or (r, par, ch) ->
   let l1 = List.map getStatus (List.map get_rec !ch) in
     if List.exists (fun x -> x = True) l1 then r.status <- True
       else r.status <- Unknown;
                      if par = None then r.status
                      else updateNodes (get_parent par)
```

If `n` is matched to an And node, we return the corresponding result.

```
| And (r, par, ch) ->
   let l1 = List.map getStatus (List.map get_rec !ch) in
     if List.exists (fun x -> x = False) l1 then r.status <- False
     else if List.for_all (fun x -> x = True) l1 then r.status <- True
     else r.status <- Unknown;
                    updateNodes (get_parent par)
```

Since the parent nodes are not Leaf nodes, the corresponding result will be as follows:

```
| Leaf (r, par) -> failwith "No such case."
```

7.3 Expanding Trees

At last, we can match the formula of the leaf node to the roots of AND/OR trees to expand the original tree.

```
let con = getContent r in
    if ((expand_root (get_root con) c) = True) then r.status <- True
          else if ((expand_root (get_root con) c) = False)
```

```
                    then r.status <- False
              else r.status <- Unknown
```

Putting all together, we get the following code for the main function.

```
let check checkExpr c = expand_root (get_root checkExpr) c
```

It simply gets the root, builds the starter tree and then combines the four phases. The whole code is given in the appendix.

CHAPTER 8

CONCLUSION

Proofs produced by a mechanized mathematics system must take relatively large inference steps, so that users are not overwhelmed with too many details. Consequently the user wants to simplify expressions to make them more understandable. Because simplification in a logic like Chiron involves large numbers of definedness requirements, it is crucial to involve a considerable amount of definedness checking.

There are many approaches to handling partial functions and undefinedness in traditional logics, but there are fewer mechanized mathematics systems developed based on a logic with undefinedness. Our idea on reasoning about definedness in mechanized mathematics systems have been greatly inspired by IMPS. Ideas like context and two levels of definedness checking are employed in it. This thesis has presented a Chiron definedness checking system for the MathScheme project that can check if the proper expressions of Chiron are well defined or defined with a value in a particular type. The most significant feature of this work is the design and implementation of an AND/OR three-based approach for automated definedness checking which is based on ideas from artificial intelligence. Another contribution of this work is to present and implement a new definedness checking mechanism.

We have provided theorems used to determining whether an expression is defined or not, and theorems concerning the relations between types. In addition, a formalization of three-valued AND/OR trees for checking problems of definedness has been

given by slightly changing the definitions of nodes for AND/OR trees. Moreover, we have proposed a new definedness checking algorithm which combine the breadth-first methods and backward search to check the definedness problems held by nodes of AND/OR trees. Finally, a full functional definedness checking system has been implemented in OCaml. This system provides automated support for reducing a definedness problem of Chiron to simpler definedness problems that can be easily, in some cases automatically, checked in the the course of proof. Although determining the definedness of an expression is an undecidable problem, based on evidence from definedness checking in IMPS, most of the definedness checking of Chiron proper expressions can likely be done automatically by a fully implemented version of this system.

CHAPTER 9

FUTURE WORK

The definedness checking system could be improved in the future in the following ways:

In the current work, we chose the simple strategy to find only one solution tree. When the definedness information of initial formula is obtained, the checking process is stopped. Thus, the first extension of the approach could be to have a mechanism to travel the whole tree (when the tree is not big) to obtain more complete definedness information.

Furthermore, the upper level of definedness checking mentioned in Section 2.4 could be considered, but regretfully that approach to call the simplifier to reduce the results of the lower level was not available at the time of writing this thesis. Therefore, in the future, another improvement to the system may be brought through additional levels of definedness checking for more a complete definedness checking activity.

Moreover, the definedness checking system should have a tracker that can store a definedness checking processes and display the processes step by step when the user wants to see it.

Finally, because this thesis only presented 13 categories of AND/OR trees based on the main categories of proper expressions of Chiron, the most ambitious plan includes

developing starter trees for all categories of Chiron proper expressions. The other categories of theorems should be applied. The other categories of proper expressions can be handled similarly as the way provided in this work.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep gratitude to my academic supervisor, Dr. William M. Farmer, for guiding me throughout the complete process of developing this thesis and implementation, and for his support and lots of valuable and irreplaceable help during my whole stay at McMaster University. He gave me many worthy suggestions on helping me formulate this thesis and realize my ideas. Without his inspiration and encouragement, it would not have been possible to complete this thesis.

I sincerely appreciate the time spent by my examination committee members, Dr. Jacques Carette and Dr. Ridha Khedri, on their valuable and detailed responses to the first draft of my thesis.

My special thanks goes to my family, my father Shun Qing Hu and mother Si Xin Luo, for raising me and always standing by me. Without their selfless love and support, I could not have been able to make it this far. I love you my mom and dad!

APPENDIX — SOURCE CODE

This appendix contains a complete Ocaml code implementation for the Chiron definedness checking system. Section 1 introduces the definition of types for AND/OR trees. Section 2 presents the method for building AND/OR trees corresponding to different categories of Chiron expressions. Section 3 and 4 show how the leaf checking procedure and node labeling procedure works. Finally, the main function of the system is given.

A.1 Type Definitions for AND/OR Trees

```
(*Define the types and operations of nodes*)

type status = Unchecked | True | False | Unknown

(*Record of nodes*)
type node_rec = {mutable content : `p formula;
                 mutable status : status}

(*Each node has node record, a single parent, and a list of children*)
type node =
```

```
  | Leaf of node_rec * node option
  | And of node_rec * node option * (node list) ref
  | Or of node_rec * node option * (node list) ref

let creat_and x = And (x, None, ref [])
let creat_Or x = Or (x, None, ref [])

let get_rec = function
    Leaf (record, _) -> record
  | And (record, _, _) -> record
  | Or (record, _, _) -> record

let get_parent = function
    Some p -> p
  | None -> failwith "No parent."

let getContent x = x.content
let getStatus x = x.status
```

A.2 Implementing Categories of Starter Trees

```
let get x = x
```

A.2.1 General Cases

```
let build_gcsDef cname t1 t2=
  let rcont  = FOpApp ("Defined_in", [KDTerm (`Constant
```

```
        (cname,TConstant (cname)), t1); KDType t2]) in
  let lcont1 = FOpApp ("Defined_in", [KDTerm (`Constant (cname,
               TConstant (cname)), t1); KDType (TConstant "Class")])
               in
  let lcont2 = FOpApp ("Type_le", [KDType t1; KDType t2]) in
  let rootc  = ref [] in
  let and1c  = ref [] in
  let root   = Or ({content = `p rcont; status = Unchecked}, None,
               rootc) in
  let and1   = And ({content = `p rcont; status = Unchecked}, Some
      root, and1c) in
  let leaf1  = Leaf ({content = `p lcont1; status = Unchecked}, Some
               and1) in
  let leaf2  = Leaf ({content = `p lcont2; status = Unchecked}, Some
               and1) in
    and1c := [leaf1; leaf2];
    rootc := [and1];
    get root;;

let build_gcsUDef cname t1 t2  =
  let rcont  = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
      [KDTerm (`Constant (cname, TConstant (cname)), t1);
      KDType t2]))]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
      [KDTerm (`Constant (cname, TConstant (cname)), t1);
      KDType (TConstant "Class")]))]) in
  let lcont2 = FOpApp ("Type_equal", [KDType t2; KDType (TConstant
               "Empty_type")]) in
  let rootc  = ref [] in
  let root   = Or ({content = `p rcont; status = Unchecked}, None,
               rootc) in
  let leaf1  = Leaf ({content = `p lcont1; status = Unchecked}, Some
               root) in
  let leaf2  = Leaf ({content = `p lcont2; status = Unchecked}, Some
               root) in
```

```
    rootc := [leaf1;leaf2];
    get root;;
```

A.2.2 Operator Application

```
let kinded_to_trm_typ = function
  | KDTerm (p,t) -> p,t
  | _ -> failwith "Nothing."

let build_tOpAppC op1 kdl =
  let rcont = FOpApp ("Type_equal", [KDType (TOpApp (op1, kdl));
     KDType (TConstant "Class")]) in
  let lcont1 = FOpApp ("Strict", [KDTerm (`Constant (op1, TConstant
              op1), TConstant op1)]) in
  let lorcont = FBinder (Exists, trm, TConstant "Class", FOpApp
       ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
       (`Constant (trm, C.type of trm), C.type of trm);
       KDType (TConstant "Class")]))])) in
  let rootc = ref [] in
  let and1c = ref [] in
  let lor1c = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
            in
  let and1 = And ({content = `p rcont; status = Unchecked},Some root,
            and1c) in
  let lor1 = Or ({content = `p lorcont; status = Unchecked},Some root,
            lor1c) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
             and1) in
  let leaf_list1 = ref [] in
  let trm_typ = List.map kinded_to_trm_typ kdl in
  let and_array = Array.make (List.length trm_typ) (And ({content = `p
```

```
  rcont; status = Unchecked}, Some root, ref [])) in
  let i = ref 0 in
    while !i < (List.length trm_typ) -1 do
      let (a,typ) = List.nth trm_typ !i in
      let leaf_list2 = [Leaf ({content = `p (FOpApp ("Defined_in",
      [KDTerm (a, C.type of a); KDType "Class"])); status =
      Unchecked}, Some (Array.get and_array(!i))); Leaf ({content = `p
      (FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (a,
      TConstant "Class"); KDType (typ)]))])); status = Unchecked},
      Some (Array.get and_array !i))] in
        Array.set and_array !i (And ({content = `p rcont; status =
        Unchecked}, Some root, ref leaf_list2));
        leaf_list1 := !leaf_list1 @ [Leaf ({content = `p (FOpApp
        ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (a, C.type
        of a); KDType "Class"]))])); status = Unchecked}, Some lor1)];
        i := !i+1
    done;
    lor1c := !leaf_list1;
    and1c := [leaf1; lor1];
    rootc := (Array.to_list and_array) @ [and1];
    get root;;

let build_opAppUDef op1 kdl =
  let rcont = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
     (`OpApp (op1, kdl, TConstant op1), C.type of (op1, kdl,
     TConstant op1)); KDType (TConstant "Class")]))]) in
  let lcont1 = FOpApp ("Strict", [KDTerm (`Constant (op1, TConstant
      op1), TConstant op1)]) in
  let lorcont = FBinder (Exists, trm, TConstant "Class", FOpApp
       ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
       (`Constant (trm, C.type of trm), C.type of trm);
       KDType (TConstant "Class")]))])) in
  let rootc = ref [] in
  let and1c = ref [] in
  let lor1c = ref [] in
```

```
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let and1 = And ({content = `p rcont; status = Unchecked},Some root,
    and1c) in
  let lor1 = Or ({content = `p lorcont; status = Unchecked},Some root,
            lor1c) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
             and1) in
  let leaf_list1 = ref [] in
  let trm_typ = List.map kinded_to_trm_typ kdl in
  let and_array = Array.make (List.length trm_typ) (And ({content = `p
  rcont; status = Unchecked}, Some root, ref [])) in
  let i = ref 0 in
    while !i < (List.length trm_typ) -1 do
      let (a,typ) = List.nth trm_typ !i in
      let leaf_list2 = [Leaf ({content = `p (FOpApp ("Defined_in",
      [KDTerm (a, C.type of a); KDType (TConstant "Class")])); status
      = Unchecked}, Some (Array.get and_array(!i))); Leaf ({content =
      `p (FOpApp ( "Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
      (a, TConstant "Class"); KDType (typ)]))])); status = Unchecked},
      Some (Array.get and_array !i))] in
        Array.set and_array !i (And ({content = `p rcont; status =
        Unchecked}, Some root, ref leaf_list2));
        leaf_list1 := !leaf_list1 @ [Leaf ({content = `p (FOpApp
        ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (a, C.type
        of a); KDType (TConstant "Class")]))])); status = Unchecked},
        Some lor1)];
        i := !i+1
    done;
    lor1c := !leaf_list1;
    and1c := [leaf1; lor1];
    rootc := (Array.to_list and_array) @ [and1];
    get root;;

let build_fOpAppFalse op1 kdl =
```

```
let rcont = FOpApp ("Formula_equal", [KDFormula (FOpApp (op1, kdl));
   KDFormula (FConstant "False")]) in
let lcont1 = FOpApp ("Strict", [KDTerm (`Constant (op1, TConstant
    op1), TConstant op1)]) in
let lorcont = FBinder (Exists, "trm", TConstant "Class", FOpApp
     ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
     (`Constant (trm, (TConstant "Expr_term")), TConstant
     "Expr_term"); KDType (TConstant "Class")]))])) in
let rootc = ref [] in
let and1c = ref [] in
let lor1c = ref [] in
let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
  in
let and1 = And ({content = `p rcont; status = Unchecked},Some root,
  and1c) in
let lor1 = Or ({content = `p lorcont; status = Unchecked},Some root,
  lor1c) in
let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
   and1) in
let leaf_list1 = ref [] in
let trm_typ = List.map kinded_to_trm_typ kdl in
let and_array = Array.make (List.length trm_typ) (And ({content = `p
rcont; status = Unchecked}, Some root, ref [])) in
let i = ref 0 in
  while !i < (List.length trm_typ) -1 do
    let (a,typ) = List.nth trm_typ !i in
    let leaf_list2 = [Leaf ({content = `p (FOpApp ("Defined_in",
    [KDTerm (a, TConstant "Expr_term"); KDType (TConstant
    "Class")])); status = Unchecked}, Some (Array.get
    and_array(!i))); Leaf ({content = `p (FOpApp ("Not", [KDFormula
    (FOpApp ("Defined_in", [KDTerm (a, TConstant "Class"); KDType
    (typ)]))])); status = Unchecked}, Some (Array.get and_array
    !i))]
    in
      Array.set and_array !i (And ({content = `p rcont; status =
```

```
        Unchecked}, Some root, ref leaf_list2));
        leaf_list1 := !leaf_list1 @ [Leaf ({content = `p (FOpApp (
        "Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (a, TConstant
        "Expr_term"); KDType (TConstant "Class")]))])); status =
        Unchecked}, Some lor1)];
        i := !i+1
    done;
    lor1c := !leaf_list1;
    and1c := [leaf1; lor1];
    rootc := (Array.to_list and_array) @ [and1];
    get root;;
```

A.2.3 Type Application

```
let build_tAppEmp t trm  =
  let rcont = FOpApp ("Type_equal", [KDType (TTypeApp (t,trm)); KDType
         (TConstant "Empty_type")]) in
  let lcont1 = FOpApp ("Type_equal", [KDType t; KDType (TConstant
      "Empty_type")]) in
  let rootc = ref [] in
  let root = Or ({content =`p rcont; status = Unchecked},None, rootc)
    in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
              root) in
    rootc := [leaf1];
    get root;;

let build_tAppC t trm  =
  let rcont = FOpApp ("Type_equal", [KDType (TTypeApp (t, trm));
     KDType (TConstant "Class")]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
```

```
              [KDTerm (trm, TConstant "Expr_term"); KDType (TConstant
      "Class")]))]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
            in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
             root) in
    rootc := [leaf1];
    get root;;
```

A.2.4 Dependent Function Types

```
let build_depFunDef t1 t2 t3 t4 trm =
  let rcont = FOpApp ("Type_le", [KDType (TBinder (Dep_fun_type, trm,
     t1, t2)); KDType (TBinder (Dep_fun_type, trm, t3, t4))])
     in
  let lcont1 = FOpApp ("Type_le", [KDType t1; KDType t3]) in
  let lcont2 = FBinder (Forall, trm, t1, FOpApp ("Type_le", [KDType
        t2; KDType t4])) in
  let rootc = ref [] in
  let and1c = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let and1 = And ({content = `p rcont; status = Unchecked}, Some root,
    and1c) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     and1) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     and1) in
    and1c := [leaf1;leaf2];
    rootc := [and1];
```

```
    get root;;

let build_depFunUDef f t1 t2 trm =
  let rcont = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
              (f, TConstant "Class"); KDType (TBinder (Dep_fun_type,
              trm, t1, t2))]))]) in
  let lcont1 = FOpApp ("And", [KDFormula (FOpApp ("Term_equal",
      [KDterm (f, TConstant "Class"); KDTerm (`Constant
      ("Empty_set", TConstant "Empty_set"), TConstant
      "Class"); KDType (TConstant "Class")])); KDFormula
      (FOpApp ("Type_equal", [KDType t1; KDType (TConstant
      "Empty_type")]))]) in
  let lcont = FOpApp ("And", [KDFormula (FOpApp ("Term_equal", [KDterm
     (f, TConstant "Class"); KDTerm (`Constant ("Empty_set",
     TConstant "Empty_set"), TConstant "Class"); KDType
     (TConstant "Class")])); KDFormula (FOpApp ("Type_equal",
     [KDType t2; KDType (TConstant "Empty_type")]))]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
             in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     root) in
    rootc := [leaf1;leaf2];
    get root;;
```

A.2.5 Function Application

```
let build_funAppDef f t1 t2 t3 trm1 trm2 =
  let rcont = FOpApp ("Defined_in", [KDTerm (`FunApp (`Constant (f,
```

```
   TBinder (Dep_fun_type, trm1, t1, t2)), trm2), `TConstant
   "Class"); KDType t3]) in
let lcont1 = FOpApp ("Defined_in", [KDTerm (trm2, `TConstant
    "Class"); KDType t1]) in
let lcont2 = FOpApp ("Defined_in", [KDTerm (`Constant (f, TBinder
    (Dep_fun_type, trm1, t1, t2)), TConstant "Class");
    KDType (TConstant "Class")]) in
let lcont3 = FBinder (Forall, trm1, t1, FOpApp ("Type_le", [KDType
    t2; KDType t3])) in
let lcont4 = FOpApp ("Defined_in", [KDTerm (`FunApp (`Constant (f,
    TConstant "Class"), trm2), `TConstant "Class"); KDType
    (TConstant "Class")]) in
let lcont5 = FBinder (Forall, trm1, t1, FOpApp ("Type_le", [KDType
    t2; KDType t3])) in
let rootc = ref [] in
let and1c = ref [] in
let and2c = ref [] in
let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
  in
let and1 = Or ({content = `p rcont; status = Unchecked},None, and1c)
  in
let and2 = Or ({content = `p rcont; status = Unchecked},None, and2c)
  in
let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
   and1) in
let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
            and1) in
let leaf3 = Leaf ({content = `p lcont3; status = Unchecked}, Some
   and1) in
let leaf4 = Leaf ({content = `p lcont4; status = Unchecked}, Some
   and2) in
let leaf5 = Leaf ({content = `p lcont5; status = Unchecked}, Some
   and2) in
  and1c := [leaf1; leaf2; leaf3];
  and2c := [leaf4; leaf5];
```

```
    rootc := [and1; and2];
    get root;;

let build_funAppUDef f t1 trm =
  let rcont = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
     ((`FunApp (`Constant (f, TConstant "Class"),trm)),
     TConstant "Class"); KDType t3]))]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
      [KDTerm (`Constant (f, C.type of f), C.type of f);
      KDType (TConstant "Class")]))]) in
  let lcont2 = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
      [KDTerm (trm, C.type of trm); KDType (TConstant
      "Class")]))]) in
  let lcont3 = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
      [KDTerm (`FunApp (`Constant (f, C.type of f), trm),
      C.type of trm); KDType (TConstant "Class")]))]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked}, None,
        rootc) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     root) in
  let leaf3 = Leaf ({content = `p lcont3; status = Unchecked}, Some
         root) in
    rootc := [leaf1;leaf2;leaf3];
    get root;;
```

A.2.6 Function Abstraction

```
let build_fAbsDef t1 t2 trm =
```

```
  let rcont = FOpApp ("Defined_in". [KDTerm (`FunAbs (trm, t1,
        `Constant ("Expr_term", t2)), TBinder (Dep_fun_type,
        trm, t1, t2)); KDType (TConstant "Class")]) in
  let lcont1 = FOpApp ("Defined_in", [KDTerm (`OpApp ("Type_to_term",
      [KDType t1], `Constant ("Expr_term", TConstant
      "Class")); KDType (TConstant "Set")]) in
  let lcont1 = FOpApp ("Defined_in", [KDTerm (`OpApp ("Type_to_term",
      [KDType t2], `Constant ("Expr_term", TConstant
      "Class")); KDType (TConstant "Set")]) in
  let rootc = ref [] in
  let and1c = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let and1 = And ({content = `p rcont; status = Unchecked}, Some root,
    and1c) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     and1) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     and1) in
    and1c := [leaf1;leaf2];
    rootc := [and1];
    get root;;
```

A.2.7 Conditional Terms

```
let build_ifConDef t1 trm1 trm2 fo =
  let t2 = C.typeof trm1 in
  let t3 = C.typeof trm2 in
  let rcont = FOpApp ("Defined_in", [KDTerm (`If (fo, trm1, trm2),
     TConstant "Class"); KDType t1]) in
  let lcont1 = FOpApp ("Formula_equal", [KDFormula fo; KDFormula
```

```
    (FConstant "True")]) in
let lcont2 = FOpApp ("Defined_in", [KDTerm (trm1, TConstant
    "Class"); KDType t2]) in
let lcont3 = FOpApp ("Type_le", [KDType t2; KDType t1]) in
let lcont4 = FOpApp ("Formula_equal", [KDFormula fo; KDFormula
    (FConstant "False")]) in
let lcont5 = FOpApp ("Defined_in", [KDTerm (trm2, TConstant
    "Class"); KDType t3]) in
let lcont6 = FOpApp ("Type_le", [KDType t3; KDType t1]) in
let rootc = ref [] in
let and1c = ref [] in
let and2c = ref [] in
let root = Or ({content = `p rcont; status = Unchecked},None,rootc)
  in
let and1 = And ({content = `p rcont; status = Unchecked}, Some root,
  and1c) in
let and2 = And ({content = `p rcont; status = Unchecked},Some root,
  and2c) in
let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
   and1) in
let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
   and1) in
let leaf3 = Leaf ({content = `p lcont3; status = Unchecked}, Some
   and1) in
let leaf4 = Leaf ({content = `p lcont4; status = Unchecked}, Some
   and2) in
let leaf5 = Leaf ({content = `p lcont5; status = Unchecked}, Some
   and2) in
let leaf6 = Leaf ({content = `p lcont6; status = Unchecked}, Some
   and2) in
  and1c := [leaf1;leaf2;leaf3];
  and2c := [leaf4;leaf5;leaf6];
  rootc := [and1;and2];
  get root;;
```

```
let build_ifConUDef t1 trm1 trm2 fo =
  let rcont = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
     (`If (fo, trm1, trm2), TConstant "Class"); KDType
     t1]))]) in
  let lcont1 = FOpApp ("Type_equal", [KDType t1; KDType (TConstant
      "Empty_type")]) in
  let lcont2 = FOpApp ("Formula_equal", [KDFormula fo; KDFormula
      (FConstant "True")]) in
  let lcont3 = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
      [KDTerm (trm1, TConstant "Expr_term"); KDType
      (TConstant "Class")]))]) in
  let lcont4 = FOpApp ("Formula_equal", [KDFormula fo; KDFormula
      (FConstant "False")]) in
  let lcont5 = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in",
      [KDTerm (trm2, TConstant "Expr_term"); KDType
      (TConstant "Class")]))]) in
  let rootc = ref [] in
  let and1c = ref [] in
  let and2c = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let and1 = And ({content = `p rcont; status = Unchecked}, Some root,
    and1c) in
  let and2 = And ({content = `p rcont; status = Unchecked}, Some root,
    and2c) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     and1) in
  let leaf3 = Leaf ({content = `p lcont3; status = Unchecked}, Some
     and1) in
  let leaf4 = Leaf ({content = `p lcont4; status = Unchecked}, Some
     and2) in
  let leaf5 = Leaf ({content = `p lcont5; status = Unchecked}, Some
     and2) in
```

```
    and1c := [leaf2;leaf3];
    and2c := [leaf4;leaf5];
    rootc := [leaf1;and1;and2];
    get root;;
```

A.2.8 Definite Description

```
let build_defDesDef t1 t2 fo trm =
  let rcont = FOpApp ("Defined_in", [KDTerm (`Binder (Def_des, trm,
     t1, fo), TConstant "Class"); KDType t2]) in
  let lcont1 = FBinder (Uni_exists, trm, t1, fo) in
  let lcont2 = FOpApp ("Type_le", [KDType t1; KDType t2]) in
  let rootc = ref [] in
  let and1c = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let and1 = And ({content = `p rcont; status = Unchecked}, Some root,
    and1c) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     and1) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     and1) in
    and1c := [leaf1;leaf2];
    rootc := [and1];
    get root;;

let build_defDesUDef t1 t2 fo trm =
  let rcont = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
     (`Binder (Def_des, trm, t1, fo), TConstant "Class");
     KDType t2]))]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FBinder (Uni_exists, trm,
```

```
      t1,fo))]) in
  let lcont2 = FOpApp ("Type_equal", [KDType t2; KDType (TConstant
      "Empty_type")]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     root) in
    rootc := [leaf1;leaf2];
    get root;;
```

A.2.9 Indefinite Description

```
let build_indefDesDef t1 t2 fo trm =
  let rcont = FOpApp ("Defined_in", [KDTerm (`Binder (Indef_des, trm,
     t1, fo), TConstant "Class"); KDType t2]) in
  let lcont1 = FBinder (Exists, trm, t1, fo) in
  let lcont2 = FOpApp ("Type_le", [KDType t1; KDType t2]) in
  let rootc = ref [] in
  let and1c = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let and1 = And ({content = `p rcont; status = Unchecked}, Some root,
    and1c) in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     and1) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     and1) in
    and1c := [leaf1;leaf2];
```

```
    rootc := [and1];
    get root;;

let build_indefDesUDef t1 t2 fo trm =
  let rcont = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
     (`Binder (Indef_des, trm, t1, fo), TConstant "Class");
     KDType t2]))]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FBinder (Exists, trm, t1,
      fo))]) in
  let lcont2 = FOpApp ("Type_equal", [KDType t2; KDType (TConstant
      "Empty_type")]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
  let leaf2 = Leaf ({content = `p lcont2; status = Unchecked}, Some
     root) in
    rootc := [leaf1;leaf2];
    get root;;
```

A.2.10 Quotation

```
let build_quoDef t1 t2 =
  let rcont = FOpApp ("Defined_in", [KDTerm (`Quote t1); KDType t2])
     in
  let lcont = FOpApp ("Type_le", [KDType (TConstant "Expr_type");
     KDType t2]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
```

```
  let leaf = Leaf ({content = `p lcont; status = Unchecked}, Some
        root) in
    rootc := [leaf];
    get root;;

let build_quoDef trm t =
  let rcont = FOpApp ("Defined_in", [KDTerm (`Quote trm); KDType t])
     in
  let lcont = FOpApp ("Type_le", [KDType (TConstant "Expr_term_type");
     KDType t]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let leaf = Leaf ({content = `p lcont; status = Unchecked}, Some
    root) in
    rootc := [leaf];
    get root;;
```

A.2.11 Evaluation

```
let build_tEvalC trm =
  let rcont = FOpApp ("Type_equal", [KDType (TEval (trm)); KDType
     (TConstant "Class")]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FOpApp ("Gea", [KDTerm (trm,
      TConstant "Class"); KDType (TConstant "Expr_type")]))])
      in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
```

```
    rootc := [leaf1];
    get root;;

let build_evalUDef trm =
  let rcont = FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
     (`Eval (trm, TConstant "Class"), TConstant "Class");
     KDType (TConstant "Expr_type")]))]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FOpApp ("Gea", [KDTerm (trm,
      TConstant "Class"); KDType (TConstant "Expr_type")]))])
      in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
    rootc := [leaf1];
    get root;;

let build_fEvalFalse trm =
  let rcont = FOpApp ("Formula_equal", [KDFormula (FEval (trm));
     KDFormula (FConstant ("False"))]) in
  let lcont1 = FOpApp ("Not", [KDFormula (FOpApp ("Gea", [KDTerm (trm,
      TConstant ("Class")); KDFormula (FConstant
      ("Expr_formula"))]))]) in
  let rootc = ref [] in
  let root = Or ({content = `p rcont; status = Unchecked},None, rootc)
    in
  let leaf1 = Leaf ({content = `p lcont1; status = Unchecked}, Some
     root) in
    rootc := [leaf1];
    get root;;
```

A.3 Building Starter Trees

```
let rec get_root checkExpr =
  match checkExpr with
    | FOpApp ("Defined_in", [KDTerm (`Constant (cname, TConstant
    (cname)), t1); KDType t2]) ->
      build_gcsDef cname t1 t2
    | FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
    (`Constant (cname, TConstant (cname)), t1); KDType t2]))]) ->
      build_gcsUDef cname t1 t2
    | FOpApp ("Type_equal", [KDType (TOpApp (op1, kdl)); KDType
    (TConstant "Class")]) ->
      build_tOpAppC op1 kdl
    | FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (`OpApp
    (op1, kdl, TConstant op1), C.type of (op1, kdl, TConstant op1));
    KDType (TConstant "Class")]))]) ->
      build_opAppUDef op1 kdl
    | FOpApp ("Formula_equal", [KDFormula (FOpApp (op1, kdl));
    KDFormula (FConstant "False")]) ->
      build_fOpAppFalse op1 kdl
    | FOpApp ("Type_equal", [KDType (TTypeApp (t,trm)); KDType
    (TConstant "Empty_type")]) ->
      build_tAppEmp t trm
    | FOpApp ("Type_equal", [KDType (TTypeApp (t, trm)); KDType
    (TConstant "Class")]) ->
      build_tAppC t trm
    | FOpApp ("Type_le", [KDType (TBinder (Dep_fun_type, trm, t1,
    t2)); KDType (TBinder (Dep_fun_type, trm, t3, t4))]) ->
      build_depFunDef t1 t2 t3 t4 trm
    | FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (f,
    TConstant "Class"); KDType (TBinder (Dep_fun_type, trm, t1,
    t2))]))]) ->
      build_depFunUDef f t1 t2 trm
```

```
| FOpApp ("Defined_in", [KDTerm (`FunApp (`Constant (f, TBinder
(Dep_fun_type, trm1, t1, t2)), trm2), `TConstant "Class"); KDType
t3]) ->
  build_funAppDef f t1 t2 t3 trm1 trm2
| FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
((`FunApp (`Constant (f, TConstant "Class"),trm)), TConstant
"Class"); KDType t3]))]) ->
  build_funAppUDef f t1 trm
| FOpApp ("Defined_in". [KDTerm (`FunAbs (trm, t1, `Constant
("Expr_term", t2)), TBinder (Dep_fun_type, trm, t1, t2)); KDType
(TConstant "Class")]) ->
  build_fAbsDef t1 t2 trm
| FOpApp ("Defined_in", [KDTerm (`If (fo, trm1, trm2), TConstant
"Class"); KDType t1]) ->
  build_ifConDef t1 trm1 trm2 fo
| FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (`If
(fo, trm1, trm2), TConstant "Class"); KDType t1]))]) ->
  build_ifConUDef t1 trm1 trm2 fo
| FOpApp ("Defined_in", [KDTerm (`Binder (Def_des, trm, t1, fo),
TConstant "Class"); KDType t2]) ->
  build_defDesDef t1 t2 fo trm
| FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
(`Binder (Def_des, trm, t1, fo), TConstant "Class"); KDType
t2]))]) ->
  build_defDesUDef t1 t2 fo trm
| FOpApp ("Defined_in", [KDTerm (`Binder (Indef_des, trm, t1, fo),
TConstant "Class"); KDType t2]) ->
  build_indefDesDef t1 t2 fo trm
| FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm
(`Binder (Indef_des, trm, t1, fo), TConstant "Class"); KDType
t2]))]) ->
  build_indefDesUDef t1 t2 fo trm
| FOpApp ("Defined_in", [KDTerm (`Quote t1); KDType t2]) ->
  build_quoDef t1 t2
| FOpApp ("Defined_in", [KDTerm (`Quote trm); KDType t]) ->
```

```
      build_quoDef trm t
    | FOpApp ("Type_equal", [KDType (TEval (trm)); KDType (TConstant
    "Class")]) ->
      build_tEvalC trm
    | FOpApp ("Not", [KDFormula (FOpApp ("Defined_in", [KDTerm (`Eval
    (trm, TConstant "Class"), TConstant "Class"); KDType (TConstant
    "Expr_type")]))]) ->
      build_evalUDef trm
    | FOpApp ("Formula_equal", [KDFormula (FEval (trm)); KDFormula
    (FConstant ("False"))]) ->
      build_fEvalFalse trm
    | _ -> failwith "Can not be checked."
```

A.4 Node Labeling Procedure

```
let rec updateNodes n =
  match n with
    | Or (r, par, ch) ->
    let l1 = List.map getStatus (List.map get_rec !ch) in
      if List.exists (fun x -> x = True) l1 then r.status <- True
        else r.status <- Unknown;
                        if par = None then r.status
                        else updateNodes (get_parent par)
    | And (r, par, ch) ->
      let l1 = List.map getStatus (List.map get_rec !ch) in
        if List.exists (fun x -> x = False) l1 then
           r.status <- False
        else if List.for_all (fun x -> x = True) l1 then
           r.status <- True
        else r.status <- Unknown;
                        updateNodes (get_parent par)
```

```
    | Leaf (r, par) -> failwith "No such case."
```

A.5 Leaf Checking Procedure

```
let rec expand_root n c =
  let q = Queue.create() in
    Queue.add (n) q;
    let rec bfs ns =
      let nr = get_rec n in
      let ns = getStatus nr in
        while ns = Unchecked do
          if Queue.length q = 0 then raise Not_found;
          let s = Queue.take q in
            match s with
              | Or (_, par, ch) ->
              for i = 0 to (List.length !ch) - 1 do
                Queue.add (List.nth !ch i) q
              done;
              bfs ns
              | And (_, par, ch) ->
                for i = 0 to (List.length !ch) - 1 do
                  Queue.add (List.nth !ch i) q
                done;
                bfs ns
              | Leaf (r, par) -> check_leaf r par c
      done
        and check_leaf r par c =
          while r.status = Unknown do
            let lc1 = getContent r in
              if inContext lc1 c then
                r.status <- True
```

```
                  else
begin
  let lc2 = simp_formula lc1 in
    r.content <- lc2;
               if (lc2 = R.falsef) then r.status <- False
                   else if (lc2 = R.truef) || (inContext lc2 c) then
                   r.status <- True
               else
                    begin
                      let con = getContent r in
                        if ((expand_root (get_root con) c) = True)
                        then r.status <- True
               else if ((expand_root (get_root con) c) = False) then
               r.status <- False
             else r.status <- Unknown
                    end
end
          done;
          bfs (updateNodes (get_parent par))
    in
    let nr = get_rec n in
      getStatus nr;;
```

A.6 Function for Definedness Checking

```
let check checkExpr c = expand_root (get_root checkExpr) c
```

BIBLIOGRAPHY

[1] Objective Caml. Home page at `http://caml.inria.fr/` (accessed July 26, 2011).

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

[3] W. M. Farmer, J. D. Guttman, and F. J. Thayer Fábrega. IMPS: An updated system description. In M. McRobbie and J. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 1996.

[4] William M. Farmer. A partial functions version of church's simple theory of types. *The Journal of Symbolic Logic*, 55(3):pp. 1269–1291, 1990.

[5] William M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64(3):211 – 240, 1993.

[6] William M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43:279 –294, November 1995.

[7] William M. Farmer. *Formalizing Undefinedness Arising in Calculus*, volume 3097 of *Lecture Notes in Computer Science*, pages 475 – 489. Springer Berlin / Heidelberg, 2004.

[8] William M. Farmer. Chiron: A multi-paradigm logic. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 1–19. University of Białystok, 2007.

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

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

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

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

[13] J. D. Guttman. A proposed interface logic for verification environments. Technical Report M91 - 19, The MITRE Corporation, 1991.

[14] Richard D. Jenks and Robert S. Sutor. *Axiom : The Scientific Computation System*. Springer-Verlag, 1992.

[15] Ambuj Mahanti and Amitava Bagchi. And/or graph heuristic search methods. *J. ACM*, 32:28–51, January 1985.

[16] Mathscheme: An integrated framework for computer algebra and computer theorem proving. Web site at `http://www.cas.mcmaster.ca/research/mathscheme/` (accessed July 26, 2011).

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

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

[19] Nils Robert Nilsson. *Problem-solving Methods in Artificial Intelligence*. McGraw-Hill computer science series. McGraw-Hill, New York, NY, 1971.

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

[21] Natara Shankar, Sam Owre, John M. Rushby, and David W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[22] The Coq Development Team. The Coq proof assistant reference manual: Version 8.3. Technical Report 38, June 2004. http://coq.inria.fr.

[23] Patrick Henry Winston. *Artificial Intelligence*. Addison - Wesley, Reading, MA, 1977.

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

[25] Han Yin Zhang. Simplification infrastructure for an implementation of the chiron logic. Master's thesis, McMaster University, 2010.