

# CONCURRENT REFINEMENT IN NUPRL

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Roderick Moten

August 1997



© Roderick Moten 1997  
ALL RIGHTS RESERVED



## CONCURRENT REFINEMENT IN NUPRL

Roderick Moten, Ph.D.

Cornell University 1997

This dissertation reports on the design, implementation, and analysis of the first parallel interactive theorem prover, called the *MP refiner*. The MP refiner is a shared memory multi-processor implementation of the inference engine of *Nuprl*. The inference engine of *Nuprl* is called the refiner. The MP refiner is implemented in the functional programming language Standard ML and is compatible with the refiner of *Nuprl* 4.1. Parallelism is provided in the MP refiner using an extension to the runtime system of the SML/NJ compiler for Standard ML. The MP refiner provides AND-parallelism and OR-parallelism expressed in terms of *concurrent tactics*. Concurrent tactics are created using the new tacticals PTHEN and PORELSLEL. The process of evaluating concurrent tactics is called *concurrent refinement*.

The MP refiner is a collection of threads operating as sequential refiners running on separate processors. Concurrent tactics exploit parallelism by spawning tactics to be evaluated by other refiner threads simultaneously.

Tests conducted with the MP refiner running on a four processor Sparc shared-memory multi-processor reveal that concurrent refinement can significantly de-

crease the elapsed time of constructing proofs interactively. The concurrent tactics constructed with course-grain tactics gave speedups slightly less than two. On the other hand, the concurrent tactics using fine-grain tactics gave speedups close to three. The granularity of a tactic depends on the amount of primitive inferences it invokes. The poor speedups of concurrent tactics constructed with course-grain tactics resulted from bus contention between the multiple processors caused by the memory management of SML/NJ.

The design of the MP refiner is based on a rigorous mathematical description of the refiner. The mathematical description includes a model of concurrent refinement. In addition, this dissertation presents the first complete rigorous presentation of the Nuprl second order substitution algorithm and the first thorough description of the various components that make up the refiner.

# Biographical Sketch

Roderick Moten was born on November 4, 1968 in Hempstead, NY. As a young boy, Rod was very adventurous. When Rod was 6, he climbed into an open elementary school window pried open by some older boys in his neighborhood. A security guard caught Rod shortly after he climbed through the window. Fortunately, the security guard let Rod go home with only a verbal reprimand. A great deal of Rod's play time consisted of playing hide-n-seek on the elementary school's roof, doing flips on school yard swings, and swimming at the public park. Although Rod was very adventurous, he was an intelligent young man. Math and history were his favorite subjects. However, when Rod reached high school, history became one of his worst subjects. In fact, the first year and half of high school, Rod had little appreciation for school. Although he did well in his classes, he had little interest in the material. He often skipped school to socialize with other confused young men. He also spent a considerable amount of time getting drunk and smoking marijuana. This behavior lasted until the middle of the tenth grade.

Rod's delinquent behavior changed after an encounter in his basement while smoking marijuana with a friend. In the midst of smoking a marijuana cigarette, Rod's ears began ringing loudly and he became greatly afraid. As a result, he cried

out, “God, please help me!”. God answered Rod and instructed him to throw away the marijuana cigarette, tell his friend to leave, and read the Holy Bible. Rod did as God commanded. A month later, Rod devoted his life to Jesus Christ. During this period, Rod spent most of his free time reading the Bible, praying, and going to church meetings. Also, he stopped smoking marijuana and drinking alcohol. Rod also became an A student and took his first computer course, BASIC. During this time, Rod developed a great appreciation for chemistry, math, and computers. However, Rod’s devotion to Jesus only lasted for a year and a half.

Late in the eleventh grade , Rod changed from a follower to Jesus to a hoodlum. Although Rod attended school regularly, he missed almost every class, except his classes in Pascal and pre-calculus. He spent the remainder of his time getting into trouble. Rod had little respect for authority. It was a small matter for Rod to defy the orders of a principal, teacher, and even his mother. Rod believed that he was invincible. As a result, Rod hung out at some of the most abominable places in Hempstead and New York City. He came home extremely late on school nights. One school night, Rod came home at 4 a.m. from a bar. One Friday night, he took off with one of his friends and did not return home until Sunday afternoon. Despite Rod’s delinquent behavior, he was still hard working. Some mornings Rod would wake up at 5 a.m. to go to a temporary job agency to find employment that lasted for one or two days. Most of the jobs he obtained consisted of menial tasks in factories and warehouses.

Although Rod desired to be a hoodlum, his teachers and guidance counselor knew Rod had potential to do much more. In the beginning of Rod’s senior year of

high school, Rod's guidance counselor suggested Rod apply to S.U.N.Y at Stony Brook. Rod heard that Stony Brook was a party school, so he decided to apply.

Rod began attending Stony Brook in the Fall of 1986. Although Rod was in college, he still had the desire to be a hoodlum. During his first semester in college, Rod spent the majority of his time getting drunk, attending parties, and doing everything except study. As a result, Rod obtained a 1.37 GPA out of 4.00. Rod received his grade report during winter break. During winter break, Rod worked in factories. The factory work was a positive experience for Rod: It showed him his future without a college education. One night, while out drinking with a friend, Rod made a resolution to work hard to do well in his courses. When school resumed, Rod followed through with his resolution.

In Rod's second semester at Stony Brook, he studied every day of the week. He understood his course work so well that he tutored others. Rod got so involved in his courses that he even talked about them when he got drunk. As a result, his friends nicknamed him "Professor Mo". (Mo is short for Moten.) This name stuck with him throughout his undergraduate career. He also used this name when he became a disc jockey at Stony Brook's radio station, WUSB-FM. Rod, along with others, started the first radio show at WUSB featuring House music. Rod's radio program, and his tenure as music director, made him popular on campus. As a result, Rod spent a lot of time attending parties. Rod perfected managing his time so that he could excel in his computer science and mathematics courses while attending as many parties as possible. He attended the majority of the campus parties, as well as some night clubs in New York City. However, his fervor for

partying partially diminished after a car accident that took the life of his grade school friend, Jonathan Brown. During Labor Day weekend of 1989, Jonathan and Rod were driving home on the highway from an after hours night club in New York City. Jon drove and Rod, the only passenger, sat in the passenger seat. Both Rod and Jon fell asleep in the car. As a result, the car veered off the highway and ran into the side of an over-pass. Rod almost died in the car accident. A bystander pulled him out of the car seconds before the car burst into flames. Jon, however, was trapped behind the steering wheel and died in the fire. Rod spent a day in the hospital after the crash. He only sustained a few lacerations above his right eye. When he came home from the hospital, Rod's cousin, Reggie Hickmon, visited him. Reggie told Rod that God spared Rod's life so that Rod could recommit himself to Jesus. Although Rod realized that he barely escaped death and punishment for his sinful lifestyle, he refused to devote his life to Jesus. However, in June of 1991, almost two years after the accident, Rod recommitted himself to Jesus. The conversion resulted from the realization that women, parties, success, and drinking could not fill the emptiness that he had within. Because of Rod's near death experience, Rod serves God with all of his heart, mind, and strength.

Before Rod's rededication to Jesus in 1991, Rod spent the summer of 1990 at Berkeley attending the Summer Mathematics Institute. The purpose of the Institute was to encourage people from underrepresented groups to pursue Ph.Ds in mathematics. At the Summer Mathematics Institute, Rod met famous mathematicians such as, Serge Lang, Leon Henkin, and Martin Davis, and participated in two six week seminars in topology and finite fields. Rod obtained invigorating

intellectual stimulation from lectures, seminars, and late night discussions with fellow students on mathematical problems they found interesting. After attending the Summer Mathematics Institute, Rod wanted to pursue a Ph.D in mathematics. However, Rod's mathematical background was insufficient to get into a Ph.D program for mathematics of his choice. Therefore, after obtaining his B.S. in Computer Science and Mathematics from S.U.N.Y. Stony Brook in 1991, Rod attended Cornell University to obtain his Ph.D in Computer Science.

Rod became a graduate student in the Computer Science Department at Cornell University in August of 1991. His first semester was very humbling. Rod underestimated the caliber of the work and students at Cornell. As a result, Rod became very insecure about his intellectual ability. During his first semester, Rod contemplated leaving Cornell at the end of the school year. His family was very supportive of his decision. However, Rod changed his mind after hearing stories from his mother and uncle about growing up poor in Alabama. Rod continued his studies at Cornell to receive a M.S. in computer science in 1994. In the same year, Rod was married to Natacha Joasil. Both Rod and Natacha were very active in the Holy Ghost Connection, a Christian fellowship for students at Cornell. Rod left Cornell on a leave of absence in 1996 for a position as a visiting professor at Colgate University. A year later, Rod received his Ph.D. from Cornell. Rod is the second person in his extended family of over 120 people to receive a Ph.D.

To Jesus Christ the Righteous and the memory of Jonathan Brown

# Acknowledgements

I thank my committee chair, Bob Constable for all of his encouragement, support, and guidance. Bob helped me understand the aesthetics of research presentation which is important for survival as an academic. I thank Greg Morrisett for his expertise in ML, introducing me to the MP runtime system, and his in-depth knowledge of programming issues. I thank Anil Nerode for our conversations on the future of the theorem proving. I thank Stuart Allen for helping clarify my ideas on the mathematical description of the refiner. Also I thank Stuart for helping me understand Nuprl substitution and informing of the undocumented research and decisions made towards the system design of Nuprl. I thank Rich Eaton for the number of hours of conversations regarding Nuprl 5 that gave me insight of the gory details of the implementation of Nuprl. I thank Paul Jackson for helping me get accustomed to Nuprl and allowing me to continually bug him with mundane questions. I thank Tim Griffin for introducing me to designing proof systems independent of their logic. I also appreciate Tim for many conversations and advice on Nuprl, automated deduction, and life. I thank John Reppy for his assistance with Standard ML of NJ. I thank Karl Crary for implementing a converter from Nuprl ML abstract syntax trees to SML/NJ abstract syntax trees. I thank Chet

Murthy for his criticism of my work. Although, it was not constructive, I appreciate having someone challenge my ideas. I thank my office mates, Lorenzo Alvisi, Bruce Hoppe, Katherine Guo, and Divakar, for moral support and great conversations. I especially appreciate Katherine Guo for loaning me her Unix and thread programming books. I especially thank Guerney Hunt, whose suggestion to read the first eight chapters of Proverbs was more beneficial than he can ever know. Special thanks to the people of God's Way Church of Jesus, Refuge Temple, and the First Baptized Church of the Apostolic Faith for their prayers and support. Special thanks to my friends from the Holy Ghost Connection Bible Study who were really inspiration to me at Cornell. I especially thank Jarvis for his wisdom and friendship. Special thanks to my family who were my primary reason for continuing with a Ph.D although they did not force me to do so. Special thanks to my mother, Alice Renfroe for her patience and endurance as a loving mother. Special thanks to my wife, Natacha, who comforted me during the most stressful moments of preparing this dissertation. I also thank Natacha for keeping me focused. Knowing that I had Natacha waiting at home for me, helped me concentrate on my dissertation work instead of goofing off surfing the Web or in idle conversations. Most of all I thank my lord, savior, and friend, Jesus Christ, who never left me or forsook me and gave me the energy, perseverance, and joy to endure the suffering required to complete this dissertation. All glory to you, Jesus.

# Table of Contents

Biographical Sketch	iii
Dedication	viii
Acknowledgements	ix
List of Tables	xiv
List of Figures	xvi
<b>I Background</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Parallelism in Interactive Theorem Provers . . . . .	3
1.2 Exploiting Parallelism in Nuprl . . . . .	5
1.3 Contributions . . . . .	9
1.4 Layout . . . . .	12
<b>2 Standard ML</b>	<b>13</b>
2.1 An Introduction to SML . . . . .	15
2.1.1 Base Values and Types . . . . .	15
2.1.2 Tuples, Records, and Lists . . . . .	16
2.1.3 Functions . . . . .	17
2.1.4 Type Abbreviations and Datatypes . . . . .	18
2.1.5 Pattern Matching . . . . .	20
2.1.6 Exceptions . . . . .	22
2.1.7 References . . . . .	23
2.1.8 Signatures and Structures . . . . .	23
2.2 The Visible Compiler . . . . .	25
2.2.1 Interactive Top Loop . . . . .	28
2.3 The Runtime System . . . . .	29

2.3.1	The MP Runtime System . . . . .	30
<b>II</b>	<b>The Refiner</b>	<b>33</b>
<b>3</b>	<b>Mathematical Description of the Nuprl Refiner</b>	<b>34</b>
3.1	Mathematical Definitions and Notation . . . . .	35
3.2	Nuprl Terms . . . . .	37
3.2.1	Substitution . . . . .	38
3.2.2	Representing Nuprl Terms using $\mathcal{T}$ . . . . .	40
3.2.3	Evaluating Terms . . . . .	40
3.3	Developing Proofs using Refinement . . . . .	42
3.3.1	Primitive Refinement . . . . .	42
3.3.2	Tactics . . . . .	43
3.4	Extraction . . . . .	47
3.5	Model of Evaluation of Tactic Refinement . . . . .	49
3.5.1	The Evaluation of Tactic Refinement Using Concurrency . . . . .	54
<b>4</b>	<b>Algorithms</b>	<b>57</b>
4.1	Substitution . . . . .	57
4.1.1	Avoiding Capture . . . . .	61
4.2	Equality . . . . .	65
4.2.1	The Nuprl Equality Algorithm . . . . .	70
<b>5</b>	<b>Components of the Refiner</b>	<b>71</b>
5.1	Term Module . . . . .	72
5.1.1	Implementing Parameters and Terms . . . . .	72
5.1.2	Term Evaluation . . . . .	77
5.2	Rule Interpreter . . . . .	80
5.2.1	Implementation of a Refinement Rule . . . . .	85
5.3	Proof Manager . . . . .	87
5.4	Refiner State . . . . .	90
<b>III</b>	<b>Concurrent Refinement</b>	<b>92</b>
<b>6</b>	<b>Implementation of Concurrent Refinement</b>	<b>93</b>
6.1	Implementation of the MP refiner . . . . .	94
6.2	Implementation of Concurrent Tacticals . . . . .	99
6.3	Non-determinism in Concurrent Refinement . . . . .	102

<b>7</b>	<b>Performance Analysis of Concurrent Refinement</b>	<b>108</b>
7.1	Cost Model of Tactic Applications . . . . .	108
7.2	Methods . . . . .	112
7.3	Results . . . . .	117
7.4	Discussion . . . . .	120
<b>8</b>	<b>Conclusion</b>	<b>126</b>
<b>A</b>	<b>Parameters</b>	<b>130</b>
<b>B</b>	<b>Level Expression Predicates, Operations, and Relations</b>	<b>132</b>
B.1	Computing the relations $\prec$ , $\preceq$ , and $\sim$ . . . . .	135
<b>C</b>	<b>Refiner Interface</b>	<b>137</b>
<b>D</b>	<b>Refinement Rule Specification Syntax</b>	<b>144</b>
<b>E</b>	<b>Preliminary Tests of the MP Refiner</b>	<b>147</b>
	<b>Bibliography</b>	<b>156</b>

# List of Tables

2.1	SML Base Types and Values . . . . .	15
4.1	Priorities of Binding Indexes . . . . .	63
7.1	Tactics on the Left Hand Side of THEN . . . . .	114
7.2	The Right Hand Side Tactics . . . . .	114
E.1	Preliminary Results of AND-parallelism (Part 1) . . . . .	149
E.2	Preliminary Results of AND-parallelism (Part 2) . . . . .	150
E.3	Preliminary Results of AND-parallelism (Part 3) . . . . .	151
E.4	Preliminary Results of OR-Parallelism (Part 1) . . . . .	152
E.5	Preliminary Results of OR-Parallelism (Part 2) . . . . .	153
E.6	Preliminary Results of OR-Parallelism (Part 3) . . . . .	154
E.7	Preliminary Results of OR-Parallelism (Part 4) . . . . .	155

# List of Figures

1.1	Proof of Stamps . . . . .	7
1.2	Proof Produced from a Validation . . . . .	8
2.1	Predefined Functions for Lists . . . . .	20
2.2	Sketch of Function for SML/NJ Evaluation . . . . .	28
2.3	MP Diagram . . . . .	31
4.1	Tree Representation of $t$ . . . . .	60
4.2	Algorithm for Detecting $s \leftrightarrow_E^+ t$ . . . . .	68
4.3	The Nuprl Equality Algorithm . . . . .	69
5.1	Implementation of Parameter Tags . . . . .	73
5.2	Implementation of Terms . . . . .	76
5.3	Pointer representation of $\text{nat}\{\}\()$ . . . . .	76
5.4	Optimize pointer representation of $\text{nat}\{\}\()$ . . . . .	77
5.5	Type of a Primitive Refinement Rule Specifications . . . . .	80
5.6	Type of Refinement Rules . . . . .	85
5.7	Function for Applying Refinement Rules . . . . .	86
5.8	ML code for Performing Tactic Refinement . . . . .	90
6.1	MP Refiner . . . . .	94
6.2	Interface of the Refinement Server . . . . .	95
6.3	Implementation of <code>allReplies</code> . . . . .	98
6.4	Implementation of <code>PTHEN</code> using <code>PTHENOnEach</code> . . . . .	100
6.5	Implementation of <code>PORELSEL</code> using <code>PFirst</code> . . . . .	101
6.6	Tactic Proof $P$ . . . . .	102
6.7	Tactic Proof $P_1$ . . . . .	103
6.8	Finished replayed proof of $P$ using less tactics . . . . .	103
6.9	Definition of Restricted References . . . . .	106
7.1	Definition of <code>ORELSELF</code> . . . . .	115
7.2	AND-parallelism Speedups . . . . .	118
7.3	OR-parallelism Speedups . . . . .	119

7.4	AND-parallelism Efficiencies . . . . .	120
7.5	OR-parallelism Efficiencies . . . . .	121

# Part I

## Background

# Chapter 1

## Introduction

In this dissertation, we investigate using parallel computing techniques to improve the throughput of interactive theorem provers. We focus on interactive theorem provers that use tactics to develop proofs.

An interactive theorem prover is a computer program that constructs the proof of a theorem with user assistance. The interactive theorem prover may require the user to supply a proof rule of the prover's logic for each step of inference. This kind of an interactive theorem prover is called a *proof checker*, such as AUTOMATH [25] and Mizar [64]. More sophisticated interactive theorem provers require users to supply programs, called *tactics*, to carry out inference. Tactics were first used with the LCF interactive proof system [31]. Almost all interactive theorem provers in existence today use tactics [22,30,24,56,27,60,40]. Tactics usually carry out multiple steps of inference and use heuristics to determine the inferences to employ. Using tactics is more than proof checking because a tactic may be the inference procedure of an automatic theorem prover. A tactic may construct an entire proof

or a portion of a proof. In any case, a tactic must use the proof rules of the base logic of the interactive theorem prover to construct a proof.

## 1.1 Parallelism in Interactive Theorem Provers

Parallel computing has been used in various ways to obtain high-performance applications. For example, many applications for the physical sciences, such as modeling of complex molecule structures [19] and 3D visualization of turbulence [52], require running on massively parallel supercomputers. Also, many of the computationally hard tasks of artificial intelligence programs, such as chess playing programs, employ parallel computing. Some applications would be infeasible without parallel computing [29]. We believe that parallelism will improve the performance of interactive theorem provers. Previous research has shown employing parallelism in automatic theorem provers has led to significant speedups. For instance, the parallel version of Otter [47], Roo [46,67], has near linear speedups for many of the tests reported on in [46]. The tests were conducted using the benchmark of theorem proving problems in [76]. The inference mechanism of Otter is based on the Knuth-Bendix decision procedure [42]. Roo runs on a shared-memory multi-processor with 24 processors. Parallel versions of SETHEO [45], PARTHEO [65] and SPTHEO [69,68], outperformed SETHEO, an automatic theorem prover for full first order logic. In addition, 75% of the tests with the Parthenon parallel theorem prover [17] based on Warren's SRI model of OR-parallelism of Prolog [74] had a linear speedup over the sequential version of the prover. The speedups were obtained using an Encore Multimax with 16 processors.

*Static Partitioning with slackness* [70] characterizes the model of parallelism employed in the above parallel theorem provers and several other parallel theorem provers [71,66,16]. Static partitioning with slackness divides the search process into three sequential phases. During the first phase, an initial segment of the search space is explored to obtain tasks that can be computed simultaneously using OR or independent AND-parallelism. During the second phase, the tasks are distributed amongst parallel processors. During the third phase, each of the parallel processors works on the tasks independently. After a processor completes a task, it reports its results to a master process that is responsible for global termination.

Like automatic theorem provers, tactic-style interactive theorem provers that use tactics can incorporate parallelism based on static partitioning with slackness. These theorem provers can be viewed as AND/OR-search. Tactics use AND-search to construct proofs using goal-directed reasoning. A goal is solved using goal-directed reasoning by decomposing the goal into more tractable subgoals. Then, each subgoal is solved independently. Afterward, a solution of the goal is constructed from the solutions of the subgoals. In theorem proving, the goals represent formulas or sequents in a logic and solutions represent proofs. Tactics are used to decompose a goal into subgoals. In addition to the subgoals, a tactic produces a *validation* when decomposing a goal. The validation constructs a proof for the goal using the proofs of the subgoals.

Tactics created with the *tactical ORELSE* [31] use OR-search to construct proofs. A tactical composes a new tactic from one or more existing tactics. Given tactics  $t$  and  $t'$ ,  $t \text{ ORELSE } t'$  is the tactic in which  $t$  is used first to construct a proof. If  $t$

fails, then  $t'$  is used to construct a proof.

## 1.2 Exploiting Parallelism in Nuprl

To investigate the effects of parallelism in interactive theorem provers, we re-implemented the inference mechanism of the Nuprl Proof Development System [22], the *refiner*, to employ AND-parallelism and OR-parallelism. Nuprl is an interactive theorem prover initially designed to exploit the proofs-as-programs paradigm [36,14,13]. The base logic of Nuprl, the Nuprl type theory [21,6], is a constructive type theory related to Martin-Löf's type theory [58,59]. In the first version of Nuprl, Nuprl 3, the Nuprl type theory was hard coded into the refiner. However, in subsequent versions of Nuprl, Nuprl 4.1 and Nuprl 4.2, the refiner is parameterized by *refinement rules*. As a result, Nuprl may be used as a work bench for implementing other theorem provers [51]. Proof rules of a logic or typing rules of a type theory are encoded into Nuprl as refinement rules. The successful application of a refinement rule to a goal produces zero or more subgoals. If no subgoals are produced, then the goal is true; otherwise, the goal is true if all of the subgoals are true. Successfully applying a refinement rule to a goal is called *primitive refinement*. The refiners implemented in Nuprl 4.1 and Nuprl 4.2 contain a specification language in which a user supplies the refinement rules.

Users construct proofs using primitive refinements indirectly with tactics [20]. A tactic is a program that performs one or more primitive refinements to produce zero or more subgoals from a goal. In addition, a tactic generates a validation to construct a proof of the goal from the proofs of the subgoals. Tactics may be built

in a modular fashion. First, simple tactics are created that employ one primitive refinement. These tactics use very simple heuristics or none at all. They can be described as follows.

1. Perform some heuristic.
2. Choose a primitive refinement rule based on the result of the heuristic.
3. Refine the goal with the primitive refinement rule.

More sophisticated tactics are built using simple tactics according to the following scheme.

1. Perform some heuristic.
2. Choose a collection of tactics to invoke based on the result of the heuristic.
3. With the aid of a tactical, create a new tactic using the collection of tactics.

A tactical is a combinator for creating new tactics from existing tactics.

4. Refine the goal with the new tactic.

Tactics are constructed in Nuprl using the functional programming language Nuprl ML. Nuprl ML is a variant of Cambridge ML [55]. The data structures and operations used to construct proofs are primitive types and values in Nuprl ML. For example, `proof`, the type of proofs, is a primitive type of Nuprl ML. Proofs are represented as trees. Figure 1.1 contains a partial proof in Nuprl of the elementary number theory theorem we call Stamps. Stamps says that every integer  $i \geq 8$  is the sum of a multiple of three and a multiple of five. Each node in the tree contains a goal. In Figure 1.1, the goals are sequents: The hypotheses appear above the

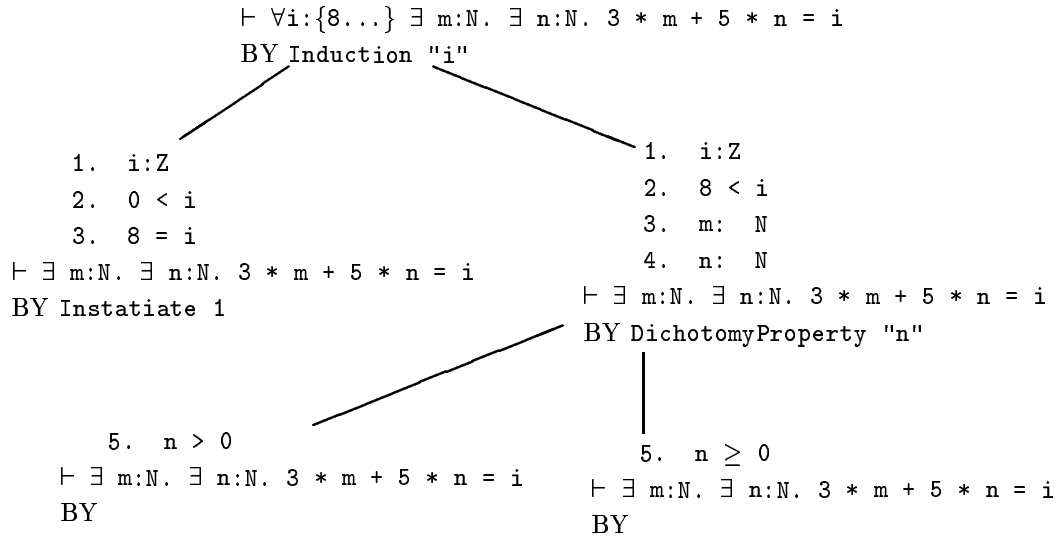


Figure 1.1: Proof of Stamps

conclusion. The text after the word BY is a *justification*. A justification is either a primitive refinement rule or a tactic. All of the justifications in Figure 1.1 are tactics. A justification provides the means of deducing the truth of a goal from its subgoals. For example, the justification Induction "i" establishes that the goal at the root of Figure 1.1 is true if the base case and inductive case for induction over  $i$  are true. The children of the root in Figure 1.1 represent the base case and the inductive case. All interior nodes of a proof contain a justification. A leaf with a justification indicates that the goal is true. The nodes without justifications represent claims that must be proven true for their parent to be true. For example, the two rightmost leaves are two claims that the inductive case is true when either  $n = 0$  or when  $n > 0$ . (See hypothesis 5 in each of the leaves.) Thus we must prove these two claims to complete the proof.

Proofs are created from a tactic using the Nuprl ML primitive function `refine`.

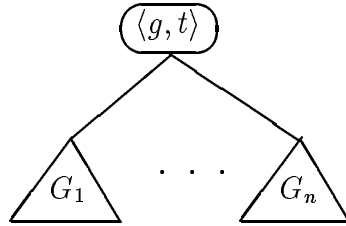


Figure 1.2: Proof Produced from a Validation

The function `refine` takes a tactic and a goal and generates a list of subgoals and a validation. Successfully applying `refine` to a tactic and a goal is called *tactic refinement*. Let  $t$  be a tactic and  $g$  be a goal. Suppose applying `refine` to  $t$  and  $g$  produces the subgoals  $g_1, \dots, g_n$  and the validation  $v$ . Then given any proofs  $G_1, \dots, G_n$  of  $g_1, \dots, g_n$  respectively,  $v$  will generate the proof displayed in Figure 1.2.

To employ parallelism in tactic refinement, we developed the *MP refiner*, the first parallel interactive theorem prover. Although the Nuprl 4.1 refiner is implemented in Lisp, we did not implement the MP refiner in a parallel version of Lisp, like Multilisp [33]. Instead, we first re-implemented the Nuprl 4.1 refiner in the functional programming language Standard ML [48,57] without parallelism. The refiner implemented in Standard ML is compatible with the refiner in version 4.1 of Nuprl. After expanding a portion of the standard library of Nuprl 4.1, we enhanced the refiner implemented in Standard ML to create the MP refiner. **MP** [50] provides parallelism in Standard ML. **MP** extends the runtime system of Standard ML so that ML threads can be mapped to OS threads that run on unique processors. The runtime system manages memory for ML programs, provides system

services, such as I/O and file system routines, manages the ML state, and handles signals and traps caused during the execution of ML code [8]. We call the runtime system using **MP** the *MP runtime system*. Although the MP runtime system deals with operating system and hardware dependent features, it is extremely portable [50]. We use the port for Sparc multi-processors.

The MP refiner runs on a shared-memory multi-processor and provides primitives for users to run tactics that can be evaluated using AND-parallelism and OR-parallelism. We call tactics that use parallelism *concurrent tactics*. Tactic refinement using concurrent tactics is called *concurrent refinement*. AND-parallelism is obtained by creating tactics using the new tactical PTHEN. PTHEN is the concurrent version of the tactical THEN introduced by LCF. Given tactics  $t$  and  $t'$ , refining  $g$  with  $t$  PTHEN  $t'$  will first refine  $g$  using  $t$  to produce the subgoals  $g_1, \dots, g_n$ . then  $t'$  will be used to refine each  $g_i$  simultaneously. OR-parallelism is obtained by creating tactics using the new tactical PORELSEL (pronounced p-or-else-l). PORELSEL is the concurrent version of ORELSE. For instance, refining  $g$  with the tactic  $t_0$  PORELSEL  $[t_1, \dots t_n]$  results in refining  $g$  with each  $t_i$  simultaneously. One of the proofs is chosen as the proof created by  $t_0$  PORELSEL  $[t_1, \dots t_n]$ .

### 1.3 Contributions

Below I list the contributions made by this dissertation.

- **First parallel interactive theorem prover.** The MP refiner is the first parallel interactive theorem prover. Although automatic theorem provers have exploited parallelism since the late eighties, until the MP refiner, in-

teractive theorem provers have not exploited parallelism. A possible reason is the short running times of tactics. Users usually apply tactics that have short execution times, between a three quarters of a second and two minutes. These tactics use simple heuristics and perform modest automation. However, sophisticated tactics, such as the Super Hardware Tac [2,1], that seek to provide automation to prevent the user from developing tedious elementary proofs interactively may execute for five to twenty minutes. Waiting five to twenty minutes or even two minutes compromises the interactive nature of an interactive theorem prover.

- **Programming language independent model for evaluating tactics**  
We define a model of evaluation of tactics independent of any programming language. The model of evaluation describes how tactics employ refinement rules to to construct proofs. Without a language independent model of evaluation of tactics, reasoning about tactics requires reasoning within the programming language in which the tactic has been written.
- **First written work on the design of the refiner.** This dissertation documents the years of collaborative research undertaken to design and implement the Nuprl refiner. The refiner evolved from several reasoning systems such as,  $\lambda$ -Prl and PL-CV [23]. We fully describe Nuprl's second order substitution with its sophisticated renaming convention. We describe the algorithm for equality of terms in the Nuprl typing theory. In addition, we give a pragmatic view of the refiner by describing the functionality of the various components that implement the refiner.

- **Mathematical description of the refiner.** We give a mathematical description of the refiner informally related to the logic independent account of theorem proving using tactics called *abstract refinement* [32] and the reflected account of the Nuprl type theory [3].
- **Implementation of a non-trivial application using the MP runtime system.** The MP refiner is the first non-trivial application that uses the MP runtime system. The MP refiner is approximately 15,000 lines of Standard ML code. As a result, the MP refiner is a good candidate for testing the ability of the MP runtime to provide parallel support for large applications.
- **Towards an implementation of a translator from Nuprl ML to Standard ML** To use the standard tactics of Nuprl 4.1 [39] in the MP refiner without rewriting them in Standard ML, we developed a translator that converts Nuprl ML abstract syntax trees to Standard ML abstract syntax trees. Rich Eaton created a function in the Nuprl 4.1 refiner that parses Nuprl ML source code and dumps the abstract syntax tree to a file as an S-expression. Karl Crary created a Standard ML program that converts the Nuprl ML abstract syntax tree into a Standard ML abstract syntax tree. To obtain a source to source translation of Nuprl ML to Standard ML, we only need to pretty print the Standard ML abstract syntax tree as Standard ML source code.

## 1.4 Layout

The remainder of Part 1 of the thesis gives background information on Standard ML (Chapter 2), and the MP runtime system (Chapter 2.3.1).

In Part 2, we discuss the refiner. Chapter 3 contains a mathematical description of the refiner and a model of evaluation of tactics independent of any programming language. In Chapter 4, we fully describe Nuprl’s sophisticated algorithm for second order substitution. Also in Chapter 4, we describe the algorithm that tests equality amongst terms in the Nuprl type theory. In Chapter 5, we describe the various components used to implement the refiner.

Part 3 covers concurrent refinement. In Chapter 6, we describe the implementation of the MP refiner and the implementation of concurrent tacticals. Also, Chapter 6 contains a section describing the effects of the non-deterministic character of concurrent tactics on proof development. Finally, in Chapter 6, we present the results of performance tests of concurrent tacticals. In Chapter 7, we analyze the performance of concurrent refinement. As part of the analysis, we give a cost model of tactic execution.

# Chapter 2

## Standard ML

Standard ML (SML) [48,57] is a functional programming language that is one of the dialects of the ML functional programming language [31]. Other dialects of ML are Cambridge ML [55] and Caml [75,44]. We chose SML for implementing the MP refiner because of its success in implementing other theorem provers, such as HOL90, LEGO, STeP [15] and Isabelle. We list some of the features of Standard ML [10,11] on which we have capitalized.

- **Flexible strong typing.** SML has a simple flexible and sound type system. Flexibility is provided through polymorphism. Type checking is performed during compilation. In addition, the type system of SML allows automatic type inference. With type inference users do not have to annotate their programs with type constraints. Strong typing helps detect logical programming errors.

- **High level programming model.** SML provides a high-level programming model through functional programming that hides low level details such as memory management and data formatting.
- **Module system.** The ML module system allows programmers to separate their code into interface specifications and their implementations. In addition, ML modules support abstraction and parameterization.
- **Pattern matching.** ML supports pattern matching. Pattern matching allows programmers to easily destruct objects and assign the various parts to variables. Pattern matching also allows programmers to write code that uses the symbolic structure of values to control flow

Using SML, programmers can design a large system and develop a prototype simultaneously. The module system, strong flexible type system, and high level programming model provide a formalism for encoding the formal specifications of system as an ML program without regard to low level programming issues.

The compiler for SML developed jointly by researchers at AT&T Research, Bell Laboratories, Princeton University, and Yale University known as SML of NJ(SML/NJ) [11], provides separate compilation [12,34]. Separate compilation gives the programmer access to the various modules containing the functions and data structures used to compile and execute SML source code. These modules are called the *visible compiler*. In Section 2.2, we describe the visible compiler in more detail. In addition to separate compilation, **MP** [50] extends the SML/NJ runtime systems to execute ML programs in parallel. We describe **MP** in more detail in

Table 2.1: SML Base Types and Values

Type	Description	Example Literals
<code>bool</code>	booleans	<code>true</code> , <code>false</code>
<code>int</code>	integers	<code>~2</code> , <code>~1</code> , <code>0</code> , <code>1</code> , <code>2</code>
<code>real</code>	real numbers	<code>5.45646</code> , <code>4.9E~3</code>
<code>string</code>	ASCII strings	<code>"Jesus is alive"</code> , <code>"peace"</code>
<code>unit</code>	one element type	<code>()</code>

Section 2.3. Before describing these unique features of SML/NJ, we give a brief introduction to SML in the following section.

## 2.1 An Introduction to SML

In this section, we acquaint the reader with SML. We expect the reader is familiar with type theory and functional programming. We adapted our introduction from the introduction of SML in [62]. For a thorough coverage of SML, we encourage the reader to read “ML for the Working Programmer” by Larry Paulson [57].

### 2.1.1 Base Values and Types

An SML program is a collection of declarations binding names to values. Running an SML program creates an environment containing the association of names to values represented by the program. Values are obtained by evaluating expressions. The base values of SML are depicted in Table 2.1. The `~` in the literals `~2`, `~1`, and `4.9E~3` represents the negative sign. The literal `()` is the only element of the

type `unit`. An example declaration binding the variable `x` to the value 7 is given below.

```
val x = 4 + 3
```

The environment produced by the above binding contains a static portion containing the assignment of the type of `int` to `x`. We summarize the static portion of the environment as the specification below.

```
val x : int
```

### 2.1.2 Tuples, Records, and Lists

In addition to the base values in Table 2.1, SML provides tuples, records, and lists. For example, the pair `("Jesus", true)` is a member of the product type `string * bool`. The symbol `*` is the product type constructor of SML. Records are labeled tuples. For example, the above pair may be expressed as the record

```
val p1 = {person = "Jesus", aliveP = true}
```

which has type

```
{person : string, aliveP : bool}
```

The order of labeled fields is insignificant, so that

```
val p2 = {aliveP = true, person = "Jesus"}
```

is the same record as `p1`. A field may be selected from a record by prefixing the label with `#`. For example, the declaration

```
val savior = #person p2
```

binds the variable `savior` to `"Jesus"`. The `#` operator functions as the  $\pi$  operator for tuples. For example, the declaration

```
val x = #3(3,7,8,99,0)
```

binds `x` to 8. In general, the expression `#i t` evaluates to the  $i$ -th element of the tuple `t`; also `t` must have at least  $i$  elements.

A list is an ordered sequence of values of the same type. For example, the list literal `[1,2,3]` is a list of integers and has the type `int list`. The declaration

```
val l = [(1,false), (2,true)]
```

is a list of pairs of integers and booleans. Thus the type specification of `l` is

```
val l : (int * bool) list
```

### 2.1.3 Functions

Functions are declared using the keyword `fun`. For example, the declaration

```
fun fact n = if n <= 0 then 1 else n * fact(n-1)
```

binds the name `fact` to the function value that computes the factorial of an integer.

The type specification of `fact` is

```
val fact : int -> int
```

Functions are called during the evaluation of application expressions. For example, `fact 5` and `fact(5)` are both application expressions which call `fact` with an argument of 5. Functions may be defined to take multiple arguments. For example, the function `monus` defined below takes two integer arguments.

```
fun monus x y = if x < y then 0 else x - y
```

The type of `monus` is

```
int -> int -> int.
```

SML allows arguments to be *curried*. In other words, the value of applying a multiple argument function to some of its arguments results in a function. For example,

```
val monus5 = monus 5
```

declares `monus5` as a function of type `int -> int`. The function `monus5` is the same as `monus` except, the variable `x` in `monus` has been replaced with `5`. Thus given any value for `n`, the value of `monus5 n` and `monus 5 n` are the same.

Functions may also be declared using the `fn` keyword. The keyword `fn` works like  $\lambda$  in the  $\lambda$ -calculus. For example, we can declare `monus` using the following declaration.

```
val monus = fn x => fn y => if x < y then 0 else x - y
```

If we were writing the declaration of `monus` in the  $\lambda$ -calculus we would write

```
val monus =  $\lambda$  x.  $\lambda$  y . if x < y then 0 else x - y
```

Local variables are defined within functions using `let`-expressions. For example, the scope of the function `aux` exists only within the definition of `sqrt`:

```
fun sqrt x = let
  fun aux r =
    if r*r > x then r-1 else aux r+1
  in
    aux 1
  end
```

## 2.1.4 Type Abbreviations and Datatypes

In addition to binding names to values and functions, SML allows programmers to bind names to types. One method of binding types uses type abbreviations. For example,

```
type point = int * int
```

binds the name `point` as the product type `int * int`. The declaration only abbreviates the type `int * int` as `point`. To create a new type, we use the *datatype* construct. For instance,

```
datatype nat = Zero | Succ of nat
```

declares a new type `nat`. The identifiers `Zero` and `Succ` are constructors of `nat`. The constructor `Zero` is a nullary constructor of `nat` and is the only base value of `nat`. All other values of `nat` are composite values: They are created using the unary constructor `Succ`. The type expression after the keyword `of` identifies the type of the argument to `Succ`. The constructor `Succ` creates a new `nat` from an existing `nat`, for example `Succ(Zero)`. SML allows programmers to parameterize datatypes to create *type constructors*. For example, the datatype below defines a collection of types of binary trees.

```
datatype 'a tree = Empty
                | Leaf of 'a
                | Node of ('a * 'a b_tree * 'a b_tree)
```

The declaration

```
type string_tree = string tree
```

binds `string_tree` to the type of binary trees containing strings.

The list type we mentioned in Section 2.1.2 is the predefined datatype

```
datatype 'a list = nil | :: of ('a * 'a list)
infix 5 ::
```

The constructor `nil` denotes the empty list. The constructor `::` represents the cons of an element and a list. The `infix` declaration specifies `::` as a right associative

```

val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val rev : 'a list -> 'a list
val @ : ('a list * 'a list) -> 'a list
infix @

```

Figure 2.1: Predefined Functions for Lists

infix operator. Thus, `5::1` represents the cons of 5 to the list 1. Because of the importance of lists, SML provides syntax for constructing list literals as we described in Section 2.1.2. In other words

```
[a1, ... an]
```

is syntactic sugar for

```
a1 :: ... :: an :: nil
```

and `[]` is syntactic sugar for `nil`.

In Figure 2.1, we list the interfaces for the predefined list functions used later in this dissertation. The function `length` returns the length of a list; the function `map` creates a new list from a list `l` by applying a function to each element in `l`; the function `rev` reverses a list; and the function `@` appends one list to the end of another. The function `@` behaves as an infix operator.

### 2.1.5 Pattern Matching

Pattern matching allows programmers to perform binding using the structure of values and control the flow of evaluation. The declarations below use pattern matching to perform binding using the structure of values.

```

val (Succ m) = n
val (head :: tail) = l
val (a, _) = p

```

The first declaration binds `m` to a `nat`. For example, if `n` is equal to `Succ (Zero)` then `m` will be bound to `Zero`. The second declaration binds `head` and `tail` to the head and tail of `l`. The third declaration binds `a` to the first element of the pair `p`. The “`_`” in the second declaration is the *wildcard*. The wildcard is only used for matching; it does not bind to anything. For example, the following code results in a compilation error.

```
val (head :: _) = l
val r = rev _
```

Programmers use pattern matching to control flow by defining functions as a collection of clauses. Using pattern matching, the predecessor function for `pred` may be defined using two clauses:

```
fun pred Zero = Zero
  | pred (Succ n) = n
```

The first clause states that if the argument of `pred` is `Zero` then the result is `Zero`. In the second clause, the variable `n` is bound to the natural number argument of `Succ` and returned as the result of `pred`. Pattern matching may be performed with values other than datatype values. For example, the definition of the factorial function below uses pattern matching.

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

Calling `fact` with `0` causes the body of the first clause to be evaluated. Calling `fact` with any other integer value causes the body of the second clause to be evaluated.

Programmers also use pattern matching for flow control in *case expressions*. The declaration below, for example, uses pattern matching in the case expression

to bind `x` to the contents of a node in a binary tree of strings.

```
val x = case tree of
  Leaf s => s
  | Node (s,_,_) => s
  | _ => ""
```

The case expression matches the expression in between the keywords `case` and `of` to the head of the clauses in the body of the case statement in the order they appear. If the expression matches the head of a clause, then the body of the clause is evaluated. Thus, if `tree` is a leaf or an interior node, `x` is bound to the contents of the node. If `tree` is empty, then `x` is bound to `""`. It is important that the wildcard is used as the head of the last clause. If we permuted the second and last clauses, then `x` binds `""` when the tree is not empty.

### 2.1.6 Exceptions

SML uses *exceptions* to signal run time errors and exceptional conditions. For example, we modify the previous declaration of `x` to raise an exception if the tree is empty.

```
val x = case tree of
  Leaf s => s
  | Node (s,_,_) => s
  | _ => raise EmptyTree
```

If `tree` is empty, the `raise` expression raises the exception `EmptyTree`. Exceptions are values of the predefined type `exn`. New exceptions may be defined using the special datatype constructor `exception`, for example:

```
exception EmptyTree
```

Programmers use pattern matching to handle raised exceptions. For example, the following function declaration gives an example of obtaining the first element of a

list of strings. The predefined list function `hd` raises the exception `Hd` when applied to an empty list.

```
fun firstEl l = hd l handle Hd => ""
```

### 2.1.7 References

In SML, variables are immutable. However, SML provides mutable variables as references to heap cells. References are values of the predefined datatype `ref`. Programmers created references using the constructor `ref`. In the declaration

```
val z = ref 8
```

`z` binds a reference to an integer cell with type `int ref`. The cell referred by a reference is updated using the infix operator `:=`:

```
val := : ('a ref * 'a) -> unit
infix :=
```

A reference is dereferenced using the function `!`:

```
val ! : 'a ref -> 'a
```

Thus, `z := 777` updates the cell referenced by `z` to 777 and

```
val y = !z
```

binds `y` to 777.

### 2.1.8 Signatures and Structures

SML provides a module system implemented using *signatures* and *structures*. A signature represents the interface for a collection of types and values. A structure is a collection of declarations that match a signature. For example, the signature declaration below defines an interface for a module for a circular buffer.

```
signature BUFFER =
sig
  type 'a buffer
  exception Empty
  exception Full

  val buffer : int -> 'a buffer
  val current : 'a buffer -> 'a
  val insert : 'a buffer -> 'a -> 'a buffer
  val delete : 'a buffer -> 'a buffer
  val next : 'a buffer -> 'a buffer
end
```

The type `'a buffer` is the family of types of circular buffers; `Empty` is raised when attempting to delete from an empty buffer; `Full` is raised when attempting to insert into a full buffer; `buffer` creates an empty buffer; `current` returns the element in the current cell; `insert` inserts an element into an empty cell; `delete` removes the element from the current ; `next` moves the buffer pointer to the next cell. The structure declaration below implements the circular buffer module by declaring the appropriate types, functions, and exceptions.

```
structure Buffer : BUFFER =
struct
  type 'a buffer = 'a list * int * int
  exception Empty
  exception Full

  fun buffer n = ([], n, 0)
  fun current (buf,_,_) = hd buf handle Hd => raise Empty
  fun insert (buf,n,m) e =
    if (n = m then raise Full else (e::buf,n, m+1))
  fun delete (e::buf,n,m) = (buf,n,m-1)
    | delete ([],n,_) = raise Empty
  fun next (e::buf,n,m) = (buf @ [e],n,m)
end
```

The buffer is implemented as a tuple  $(l, n, m)$ , where  $l$  is a list containing the buffer elements,  $n$  is the size of the buffer, and  $m$  is the number of occupied cells. The head of the list is the element in the current cell. The portion of the declaration `:BUFFER` forces the compiler to ensure that the structure definition matches `BUFFER`: `Buffer` must contain declarations of values that have the same types as the corresponding values in `BUFFER`. Also, `Buffer` must declare types and exceptions with the same names as those in `BUFFER`.

Programmers access the objects declared by a structure by prefixing the object with the name of the structure and a period. For example, the declaration

```
val b = Buffer.buffer 7
```

binds `b` to an empty buffer with seven cells, and the declaration

```
val int_buf = int Buffer.buffer
```

binds `int_buf` as an abbreviation of the type of circular integer buffers. The declaration

```
val b1 = Buffer.insert b 3
```

binds `b1` to the buffer with 3 in the current cell.

## 2.2 The Visible Compiler

SML/NJ supports *meta-programming* with the visible compiler. Meta-programming is developing an application using a compilation system and embedding the compilation system within the application. In other words, an application compiled by SML/NJ may employ routines of SML/NJ to perform compilation and execution. SML/NJ exports modules that represents the data structures for performing compilation and execution. With the visible compiler, for example, one

may easily extend SML/NJ to interpreter an additional dialect of ML say, Caml. The programmer only needs to develop a translator from Caml abstract syntax trees to SML abstract syntax trees. Then the programmer applies various functions of the visible compiler on the SML abstract syntax trees to perform type checking, code generation, and execution. We actually employed this technique to permit the SML refiner to interpret Nuprl ML programs. Rich Eaton created a function in the Nuprl 4.1 refiner that parses Nuprl ML source code and dumps the abstract syntax tree as a Lisp s-expression to a file. Karl Crary developed an SML function that reads a file containing a Nuprl ML abstract syntax tree and converts it into an SML abstract syntax tree. We developed code using the visible compiler to type check an SML abstract syntax tree, compile it into machine code, and execute the machine code. To do a source to source translation from Nuprl ML to SML, we only need to pretty print the SML abstract syntax tree as SML source code. Dave Macqueen is currently implementing a pretty printer from SML abstract syntax trees to SML source code as a part of the SML/NJ compiler.

The visible compiler follows the ML compilation model [12]. In the ML compilation model, ML source code is evaluated with respect to an *environment*. An environment maps names to types and values.

$$\text{env} : \text{name} \rightarrow \text{type} \times \text{value}$$

The result of evaluation of source code is a new binding of names to types and values, *i.e.* a new environment.

$$\text{evaluate} : \text{source} \times \text{env} \rightarrow \text{env}$$

An environment is composed of a *static environment* and a *dynamic environment*. The static environment is a mapping from names to types. In addition, the static environment converts a name into a *persistent identifier*, abbreviated *pid*, internal names used only within the compiler. Persistent identifiers are a more concrete representation of names than strings. The dynamic environment maps each persistent identifier to the value it binds.

$$\begin{aligned} \text{env} &= \text{statenv} \times \text{dynenv} \\ \text{statenv} : & \quad \text{name} \rightarrow \text{type} \times \text{pid} \\ \text{dynenv} : & \quad \text{pid} \rightarrow \text{value} \end{aligned}$$

Evaluation consists of two phases, compilation and execution.

$$\begin{aligned} \text{compile} : & \quad \text{source} \times \text{statenv} \rightarrow \text{statenv} \times \text{codeUnit} \\ \text{execute} : & \quad \text{codeUnit} \times \text{dynenv} \rightarrow \text{dynenv} \end{aligned}$$

Compilation parses source code, performs type checking, and generates machine code. The outcome of compilation is a *compiled unit*.

$$\text{compiledUnit} = \text{statenv} \times \text{codeUnit}$$

A compiled unit contains the static environment produced during type checking and a *code unit*.

$$\text{codeUnit} = \text{code} \times \text{imports} \times \text{exports}$$

A code unit contains machine code, *imports*, and *exports*. Imports are a list of pids of free variables within the source code. Exports are a list of pids of variables being declared within the source code. Execution executes the code contained in a code

```

codeallowpagebreak
fun evaluated (src, evn) let
  val (statEnv, dynEvn) = env
  val (statEnv', codeUnit) = compile (src, statEnv)
  val dynEnv' = executed (codeUnit, statEnv)
in
  (statEnv', dynEnv')
end

```

Figure 2.2: Sketch of Function for SML/NJ Evaluation

unit. Execution uses a dynamic environment to instantiate the values associated with the imports in the machine code. The outcome of execution is a dynamic environment that maps each of the exports to values.

Using the above specifications, we sketch the function for evaluation. Using these definitions, we can see how evaluation can be composed of `compile` and `execute`. Moreover, we can perform several compilations and postpone the executions to another time.

### 2.2.1 Interactive Top Loop

The SML/NJ interactive top loop uses the function `evaluation` in Figure 2.2 to evaluate a declaration entered by the user. Evaluation performed with respect to two global environments, the *top level environment* and the *pervasive environment*. The top level environment contains the values and type information of bindings created from declarations previously evaluated. The pervasive environment contains values and type information of bindings of SML primitives. The new environment produced from evaluating a declaration is combined with the top level environment to produce a new top level environment. The pervasive remains unchanged. We

sketch the function used to implement the top level loop below.

```
fun interact () = let
  val src = readSrc ()
  val evalEnv = layer(topLevelEnv, pervEnv)
  val env = evaluate (src, evalEnv)
  val topLevelEnv = layer(env, topLevelEnv)
```

Programmers may create their own environments and use them as the top level and pervasive environments of the interactive top loop.

## 2.3 The Runtime System

The runtime system manages memory for ML programs, provides system services, such as I/O and file system routines, manages the ML state, and handles signals and traps caused during the execution of ML code [8]. The ML state is a data structure for containing the state of ML computation. The ML state is analogous to a process control block.

The runtime system can be divided into three parts, the kernel, garbage collector, and libraries. The libraries implement the various system services of the runtime system. The garbage collector handles memory management. The kernel manages the ML state, manages switching from executing ML code to provide garbage collection, system services, and handling signals and traps.

The SML/NJ runtime system uses a multi-generational garbage collector [7, 9,63]. In version 0.93 of SML/NJ, the garbage collector contained only two generations. However, in the latest version of SML/NJ, version 1.09.10, the garbage collector can contain up to 14 generations [63]. In version 1.09.10 of SML/NJ, the heap is partitioned into an allocation area and various generations. Objects are

allocated memory initially from the allocation area. The generations store older objects. The first generation contains the youngest of the old objects. Garbage collection is performed in two phases, *minor collection* and *major collection*. Minor collection is garbage collection of the allocation area. After the completion of each minor collection, a check is performed to determine the oldest generation to collect. The oldest generation that requires collection, is called the *currently collected generation*. Once the currently collected generation is determined, major collection begins. Major collection involves garbage collection of all generations up to and including the currently collected generation.

### 2.3.1 The MP Runtime System

**MP** [50] is a generic interface to a shared-memory multiprocessor for SML/NJ. **MP** extends the runtime system of SML/NJ to support multiplexing ML threads on top of threads provided by the operating system (OS threads). An ML thread is the computation unit of an ML expression.<sup>1</sup> ML threads run on a *virtual ML processor*. An example multi-threaded ML program may consist of several ML threads that run on a virtual ML processor. CML uses this model to implement concurrency [61]. To provide parallelism in SML/NJ, we use several virtual processors. Each virtual processor is executed by an OS thread that runs on a separate processor (see Figure 2.3). By having multiple virtual processors each mapped to an OS thread that can be run on different processors, we can have parallel computation within ML.

The interface of **MP** provides a language level view of an OS thread as a *proc*.

---

<sup>1</sup>ML threads are represented as continuations [50,61,73].

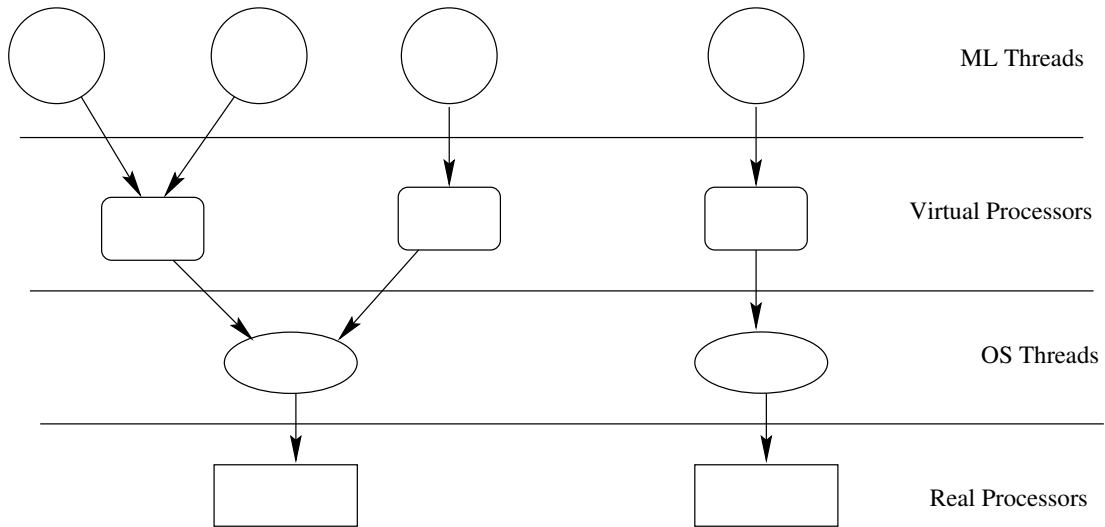


Figure 2.3: MP Diagram

All of the procs share the same heap. However the allocation space of the heap is partitioned into chunks, and each proc is given a chunk to use as a private allocation space. Although the allocation space of the heap is partitioned, garbage collection occurs over the entire heap. In addition, garbage collection is sequential: Only one proc performs garbage collection while the others wait. When a proc wants to perform a garbage collection it notifies the other procs to suspend themselves. After all of the other procs are suspended, the proc initiating the garbage collection performs garbage collection over the entire heap. When the garbage collection is completed, all of the other procs are resumed. According to [50], the cost of synchronizing for garbage collection has little effect on performance.

To facilitate partitioning the allocation space, the **MP** runtime system increases the total size of the allocation space to  $n * AllocSize$  where *AllocSize* is the size of allocation region for one proc. Therefore, a concurrent ML expression has  $n$

times as much memory for allocation than its sequential variant. As a result, the concurrent expression may require less garbage collections than the sequential expression. Hence, a concurrent expression may out perform a sequential expression because the concurrent expression spends less time performing garbage collections.

The implementation of **MP** is partitioned into a system–generic layer and a system–dependent layer. The generic layer primarily handles garbage collection and the extension to the runtime system to support multiple threads. The dependent layer primarily provides the mapping of procs to OS threads and synchronization mechanisms that can be used by the procs. The generic layer of **MP** for version 1.09.10 of SML/NJ was implemented by John Reppy and Lorenz Huelsbergen of Bell Laboratories. Huelsbergen also implemented a portion of the dependent layer responsible for synchronizing procs for garbage collection. I implemented the mapping of procs to OS threads. In particular, I implemented procs as Solaris user level threads bound to a unique light weight processes. To support parallelism, each light weight process is assigned to only run on one specific processor. Other than the benchmarks used in [50], we know of no other application than the MP refiner using **MP**.

## **Part II**

### **The Refiner**

## Chapter 3

# Mathematical Description of the Nuprl Refiner

The refiner is the component of Nuprl that performs inference. In addition to performing inference, the refiner implements the Nuprl term language. Nuprl terms resemble S-expressions in Lisp and tags in HTML: They are the syntactic entity for constructing various data objects in Nuprl.

In this chapter we provide a mathematical description of the refiner. We present the Nuprl term language as a second order language in which second order substitution can be easily expressed. In addition, we give a logic independent definition of term evaluation. Normally, term evaluation is defined with respect to the Nuprl type theory [22]. Also, we give a logic independent description of proof development based on primitive refinement and tactic refinement. Primitive refinement is the process of carrying out primitive inference. Tactic refinement is the process of carrying out multiple steps of primitive inference using a single step of inference.

We also give a formal model describing the evaluation of tactic refinement independent of any programming language. The model treats the refiner as an abstract state machine: The state is a proof, primitive refinement rules are instructions, and tactics are programs that are compiled into a list of instructions. The refiner executes the instructions to manipulate the state. Before describing the refiner, we give the mathematical definitions and notations that we use through out the dissertation.

### 3.1 Mathematical Definitions and Notation

We use type theory informally to express concepts mathematically. Our type theory has base types for all the primitive objects that are common in computer science, such as natural numbers, integers, and strings as well as the unit type of only one element. We can create new types from existing types using the well known operators  $+$ ,  $\times$ , and  $\longrightarrow$ . For type  $A$  and  $n > 0$ ,  $A^n$  denotes  $A \times A \times \dots \times A$ . We also have the  $\Pi$  and  $\Sigma$  type operators for creating product and sum types. The operator  $*$  is used to create list types. For type  $A$ ,  $A^*$  is the type inhabited with list of members of  $A$ . A list of  $n$  members of  $A$  is denoted as  $[a_1, \dots, a_n]$ . When  $n = 0$ , we denote the list as  $[\ ]$ . A list is appended to the end of another list of the same type using the operator  $\hat{\ }.$  When one of the operands of  $\hat{\ }$  contains only one element, we drop the brackets around the element. In other words, we write  $[a] \hat{\ } [b_1, \dots, b_n]$  and  $[b_1, \dots, b_n] \hat{\ } [a]$  as  $a \hat{\ } [b_1, \dots, b_n]$  and  $[b_1, \dots, b_n] \hat{\ } a$ , respectively. We say that a list  $u$  is a prefix of a list  $v$ , denoted  $u \preceq v$  if there exists a list  $w$  such that  $u \hat{\ } w = v$ . Type membership is denoted using  $\in$ . We overload  $\in$  to also denote

membership of a list.

Given a type  $T$ , we denote  $T$  tree as the type of trees whose nodes are members of  $T$ . Following the presentation of trees in [37], we identify each node of a tree using a *tree address*. A tree address is represented as a list of positive integers. We define the addresses of the nodes of a tree  $M$  as follows. The address of the root of  $M$  is the empty list. If the address of a node is  $u$  and it has  $n$  children then the address of the  $i$ -th child is  $u \hat{\ } i$ . We denote the list of addresses of all the nodes of a tree  $M$  as  $\mathcal{A}(M)$ . We use the operator  $\nu$  to obtain the node of a tree. Given a tree  $M$  and an address  $u$  in  $M$ , then  $\nu(M, u)$  is the node in  $M$  at address  $n$ . We abbreviate  $\nu(M, u)$  as  $M(u)$ .

Using tree addresses, we can easily express the replacement of a leaf of a tree with another tree as follows. Let  $M$  and  $N$  be trees and let  $u \in \mathcal{A}(M)$  that refers to a leaf. Then  $M[u \leftarrow N]$  is the tree in which  $M(u)$  is replaced with  $N$ . In other words,  $M[u \leftarrow N]$  is the tree  $M'$  where

$$M'(v) = \begin{cases} N(v') & \text{if } v = u \hat{\ } v' \\ M(v) & \text{otherwise} \end{cases}$$

Suppose  $u_1, \dots, u_k \in \mathcal{A}(M)$  and each  $u_1, \dots, u_k$  is the address of a distinct leaf in  $M$ . Then we denote the simultaneous replacement of the leaves at addresses  $u_1, \dots, u_k$  with the trees  $N_1, \dots, N_k$ , respectively, as

$$M[u_1 \leftarrow N_1 | \dots | u_m \leftarrow N_m].$$

Note that

$$M[u_1 \leftarrow N_1 | \dots | u_m \leftarrow N_m] = ((M[u_1 \leftarrow N_1])[u_2 \leftarrow N_2]) \dots [u_m \leftarrow N_m].$$

## 3.2 Nuprl Terms

We represent Nuprl terms as a subtype of the type  $\mathcal{T}$ . We defined  $\mathcal{T}$  to obtain a concise definition of second order substitution similar to the definition of second order substitution in [38] for higher order logic [18]. In addition, we defined  $\mathcal{T}$  to adequately represent the binding structure of Nuprl terms [21].

To begin, we assume that we have a countable infinite type of identifiers,  $\mathcal{I}$ , and parameters,  $\mathcal{M}$ . Parameters represent primitive objects from the underlying mathematics that we use to construct terms. Parameters are written as  $x : \alpha$ , where  $x$  is the value of the parameter and  $\alpha$  is the type of the parameter. (For more information on parameters see Appendix A.) An *operator* is an identifier paired with a list of parameters, possibly empty. The type of operators,  $\mathcal{O}$ , is the type  $\mathcal{I} \times \mathcal{M}^*$ . We write members of  $\mathcal{O}$ , for example  $\langle f, [m_1, \dots, m_n] \rangle$ , as  $f\{m_1, \dots, m_n\}$ ; If  $n = 0$ , we only use the identifier, for example  $f$ . We define the type of terms  $\mathcal{T}$  inductively as follows.

### Definition 3.2.1

1. If  $a \in \mathcal{O}$ , then  $a \in \mathcal{T}$ .
2. If  $x \in \mathcal{I}$  and  $t \in \mathcal{T}$ , then  $x.t \in \mathcal{T}$ .
3. For  $n \geq 1$ , if  $t_1, \dots, t_n \in \mathcal{T}$  and  $a \in \mathcal{O}$  then  $a(t_1; \dots; t_n) \in \mathcal{T}$ .

Members of  $\mathcal{T}$  of the form  $x.t$  are called *bound terms*:  $x$  represents a binding of  $t$ .

We abbreviate nested bound terms of the form  $t = x_1.x_2. \dots .x_n.t$  as  $x_1, x_2, \dots, x_n.t$ .

Also, if  $a \in \mathcal{T}$  and  $a \in \mathcal{O}$ , then we write it as  $a()$ .

The variables of  $\mathcal{T}$  are the members of  $\mathcal{T}$  which are variable operators. In other words, the members of  $\mathcal{T}$  of the form  $\text{variable}\{x : \mathbf{v}\}()$  are first order variables, where  $x$  is a member of the type of variable symbols  $\mathbf{v}$ . We abbreviate variables to their variable symbol. For example,  $\text{variable}\{x : \mathbf{v}\}()$  is written as  $x$ . We fix  $\mathcal{V}$  to be the type of first order variables of  $\mathcal{T}$ .

In higher order logic [18], we can detect the occurrence of a higher order variable if it occurs as a function in an application expression. For example,  $f$  in the expression  $f(a)$  is a higher order variable. In higher order logic, types are used to determine the order of a variable. The language represented by  $\mathcal{T}$  is untyped. Thus, we rely on structure to indicate the occurrence of a second order variable. As a result, *second order variable instances* are terms of the form  $\text{variable}\{a : \mathbf{v}\}(t_1; \dots; t_n)$  where  $n > 0$  and each  $t_i$  is not a bound term. The value of  $n$  is the *arity* of the second order variable instance. We abbreviate second order variable instances to only contain their variable symbol and their arguments. Thus we write  $\text{variable}\{a : \mathbf{v}\}(t_1; \dots; t_n)$  as  $a[t_1, \dots, t_n]$ .

The *free variables* of a term  $t$  is a list of variables denoted as  $FV(t)$ . If  $t \in \mathcal{V}$  then  $FV(t) = [t]$ . If  $t = a(t_1; \dots; t_n)$ , then  $FV(t) = FV(t_1) \hat{\ } \dots \hat{\ } FV(t_n)$ . If  $t = x.t'$ , then  $FV(t)$  is equal to  $FV(t')$  with all occurrences of  $x$  removed. As usual,  $t$  and  $t'$  are alpha equal,  $t =_\alpha t'$ , if they are equal up to bound variable renaming.

### 3.2.1 Substitution

We define substitution for first order variables and second order variable instances. Any term except bound terms may replace first order variables. However, only a

bound term may replace a second order variable instance.

A *substitution pair* is a pair  $\langle t, x \rangle$ , where  $x \in \mathcal{V}$  and  $t \in \mathcal{T}$ . We call  $x$  the *target* and  $t$  the *replacement*. A *substitution list* is a list of substitution pairs. We write a substitution list  $[\langle x_1, t_1 \rangle, \dots, \langle x_n, t_n \rangle]$  as  $[t_1/x_1, \dots, t_n/x_n]$ .

We write  $\sigma t$  to denote the substitution of free first order variables and second order variable instances in  $t$  with respect to the substitution list  $\sigma$ . For example,  $[A()/x, B()/v](v.(x;v))$  yields the term  $v.(A());v$ . The substitution of a second order variable instance with bound terms resembles the substitution of a second order variable with a lambda abstraction in higher order logic [38]. For example, suppose we have the second order variable instance  $f[x, y]$ . Then the result of the substitution  $[u, v.A(u;v)/f](f[B(x), y])$  is the term  $A(B(x); y)$ . We define substitution formally below.

**Definition 3.2.2** Let  $t \in \mathcal{T}$  and let  $\sigma$  be a substitution list. Also, let the free variables of the replacements of  $\sigma$  be distinct from the bound variables in  $t$ . Then  $\sigma t$  is the term obtained via substitution as follows.

1. If  $t \in \mathcal{O}, \exists \langle t, t' \rangle \in \sigma$ , and  $t'$  is not a bound term, then  $\sigma t = t'$ .
2. If  $t$  is a second order variable instance, *i.e.*  $t = f[t_1; \dots; t_n], \langle f, x_1 \dots x_n.t' \rangle \in \sigma$ , and  $t'$  is not a bound term then  $\sigma t = [\sigma t_1/x_1, \dots, \sigma t_n/x_n]t'$ .
3. If  $t$  is not a second order variable instance and  $t = a(t_1; \dots; t_n)$ , then  $\sigma t = a(\sigma t_1; \dots; \sigma t_n)$ .
4. If  $t = x_1, \dots, x_n.t'$  then  $\sigma t = x_1, \dots, x_n.\sigma' t'$ , where  $\sigma'$  is  $\sigma$  with pairs  $\langle x_i, s \rangle$  for  $i = 1, \dots, n$  removed if  $s$  is not a bound term.

5. If none of the above hold then  $\sigma t = t$ .

In the second part of Definition 3.2.2, the condition that  $t'$  is not a bound term ensures that the arity of the second order variable instance is equal to the number of bindings of the replacement. The fourth part of Definition 3.2.2 permits substituting for a second order variable instance that occurs in the scope of a binding with the same name.

### 3.2.2 Representing Nuprl Terms using $\mathcal{T}$

The Nuprl term language is represented by the subtype  $\mathcal{T}_\nu$  of  $\mathcal{T}$ : The members of  $\mathcal{T}_\nu$  are the members of  $\mathcal{T}$  that only contain bound terms as proper subterms.

**Definition 3.2.3** [Nuprl Terms] The terms of the Nuprl term language are the members of the subtype  $\mathcal{T}_\nu$  of  $\mathcal{T}$  such that  $t \in \mathcal{T}$  and  $t \neq x_1, \dots, x_n.t'$  for  $n > 0$  if and only if  $t \in \mathcal{T}_\nu$ .

Substitution for the Nuprl term language is the same as substitution for  $\mathcal{T}$ . In fact, substitution is closed for Nuprl terms. We state this as the following theorem.

**Theorem 3.2.4** Let  $\sigma$  be a substitution list and let  $t \in \mathcal{T}_\nu$ . Then  $\sigma t \in \mathcal{T}_\nu$ .

### 3.2.3 Evaluating Terms

We generalize the definition of Nuprl defined in [22,21]. The definition of term evaluation partitions terms into canonical and non-canonical terms. The evaluation procedure reduced non-canonical terms to canonical terms. For each non-canonical term, a description identifies the *principal arguments* of the term. The evaluation procedure uses a collection of unique redexes each associated with a contractum.

Evaluation of  $t$  begins by determining whether  $t$  is a canonical or non-canonical term. If  $t$  is canonical then the result of the evaluation is  $t$ . Otherwise, each of the principal arguments of  $t$  are evaluated. If each principal argument has a value, then each of the principal arguments are replaced with its respective value to produce the term  $t'$ . If  $t'$  is a redex, then evaluation continues with the contractum; otherwise,  $t$  has no value. For example, when evaluating the term

$$\text{ap}(\text{lambda}(x.\text{pair}(x; a)); b) \tag{3.1}$$

we first determine from the Nuprl type theory that it is a non-canonical term and its principle arguments are

$$\text{lambda}(x.\text{pair}(x; a)) \text{ and } b. \tag{3.2}$$

Because the terms in (3.2) are both canonical terms, they evaluate to themselves. In the Nuprl type theory, (3.1) is a redex with contractum  $\text{pair}(b; a)$ . Because  $\text{pair}(b; a)$  is a canonical term, the value of (3.1) is  $\text{pair}(b; a)$ .

We generalize evaluation by eliminating the partition of terms into canonical and non-canonical terms. Instead, the evaluation procedure is parameterized by a collection of rewrite rules, a collection of pairs of redexes and contracta, and a collection of terms paired with their principal arguments. We denote the collection of rewrite rules as the type  $\Delta$ , the collection of pairs of redexes and contracta as the type  $\Theta$ , and the collection of terms and principal arguments as the type  $\Omega$ . The evaluation procedure proceeds on a term  $t$  as follows. If  $t$  is the left hand side of a rewrite rule, then continue the evaluation on the right hand side of the rewrite rule. Otherwise, if for all lists  $l$ ,  $\langle t, l \rangle \notin \Omega$ , then the value of  $t$  is  $t$ .

However, if  $\langle t, [a_1, \dots, a_n] \rangle \in \Omega$ , then evaluate each  $a_i$  and replace them in  $t$  with their respective values to obtain the term  $t'$ . If  $\langle t', c \rangle \in \Theta$ , *i.e.*  $t'$  is a redex, then continue the evaluation with  $c$ . However, if  $t'$  is not a redex, then  $t$  has no value.

### 3.3 Developing Proofs using Refinement

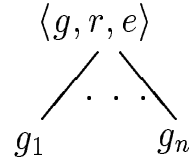
#### 3.3.1 Primitive Refinement

The refiner performs inference as *primitive refinement*. Primitive refinement is the process of applying a *refinement rule* to a *goal*. Goals are syntactic objects of the refiner used to represent some form of judgment. The user defines the semantics of a goal. For example, goals are interpreted as typing judgments in the Nuprl type theory [6]. A goal is a list  $[\langle x_1, H_1 \rangle, \dots, \langle x_n, H_n \rangle]$  paired with a Nuprl term  $C$  where for  $i = 1, \dots, n$ ,  $x_i \in \mathcal{V}$  and  $H_i \in \mathcal{T}_\nu$ . Furthermore, each free variable of  $H_i$  is some  $x_j$  for  $j < i$ . We refer to a  $\langle x_i, H_i \rangle$  as a *declaration*, *assumption*, or *hypothesis*.

A refinement rule is a partial function that maps a goal into a list of goals and a *partial extract*. (We define partial extracts in Section 3.4.) Refinement rules represent the most primitive form of inference in Nuprl. A refinement rule *refines* a goal if the refinement rule is defined on the goal. The process of refining a goal using a refinement rule is called *primitive refinement*. Using primitive refinement we construct *primitive proofs*.

**Definition 3.3.1** A primitive proof is a tree whose nodes are goals or triples  $\langle g, r, e \rangle$  where  $g$  is a goal,  $r$  is a refinement rule, and  $e$  is a partial extract. Primitive proofs are constructed using primitive refinement as follows.

1. If  $g$  is a goal then  $g$  is a primitive proof.
2. Let  $G$  be a primitive proof with a leaf node  $g$  and let  $r$  be a refinement rule. If  $r(g) = \langle [g_1, \dots, g_n], e \rangle$ , for  $n \geq 0$ , then  $G'$  is the primitive proof obtained by replacing  $g$  in  $G$  with the tree below.



The creation of a primitive proof using primitive refinement resembles the method of construction of tactic trees in [32]. In [32], tactic trees represent data structures for maintaining successive refinements.

According to Definition 3.3.1, all interior nodes are triples. Also, for any node  $\langle g, r, e \rangle$ , the goals in its children are subgoals produced by refining  $g$  by  $r$ . This criteria matches the criteria in [3] to define valid proofs in Nuprl.

A primitive proof  $G$  with  $g$  in its root is a *complete primitive proof* of  $g$  if all goals are refined in  $G$ . If  $G$  is complete all leaves of  $G$  are triples.

### 3.3.2 Tactics

A tactic is a program that constructs a primitive proof for a goal. When a tactic is invoked on a goal, it performs the necessary primitive refinements to construct a primitive proof of the goal; the proof may be incomplete. Often, a tactic employs a heuristic to perform the refinements. Tactics have no other means of constructing a proof, such as low level operations for constructing trees, other than primitive refinement. Thus, tactics cannot create invalid primitive proofs. In other words, for a tactic  $t$  and a goal  $g$ ,  $t(g)$  may be a primitive proof in which the goal contained

in the root of  $t(g)$  is not equal to  $g$ . Thus  $t(g)$  is not a primitive proof of  $g$ . To prevent the above situation, we define tactics formally as follows.

**Definition 3.3.2** Let  $t$  be a partial function that maps a goal to a list of goals and a partial extract. Then  $t$  is a tactic if  $t(g) = \langle [g_1, \dots, g_n], e \rangle$ , for  $n \geq 0$ , and there is a refinement proof  $G$  such that  $g_1, \dots, g_n$  are the leaves of  $G$ ,  $g$  is contained in the root of  $G$ , and  $e$  is the partial extract obtained by composing all the partial extracts of  $G$  using the construction defined in Definition 3.4.4

We describe two methods of constructing tactics. One method constructs tactics based on a collection of primitive refinement proofs. The other method constructs tactics using combinators called *tacticals*. In practice, users construct tactics with a functional programming language, such as ML.

Below, we define construction of *simple tactics*.

**Definition 3.3.3** Let  $A$  be a type containing pairs of goals and primitive proofs such that, if  $\langle g, G \rangle \in A$ , then  $g$  is contained in the root of  $G$ . Then  $t_A$  is the simple tactic based on  $A$  where  $t(g) = \langle [g_1, \dots, g_n], e \rangle$  if and only if  $\langle g, G \rangle \in A$ ,  $g_1, \dots, g_n$  are the leaves of  $G$ , and  $e$  is the partial extract obtained by composing all the partial extracts of  $G$  using the construction defined in Definition 3.4.4.

Our second method of constructing tactics uses *tacticals*. A tactical is a combinator for tactics. Tactics created using tacticals are called *composite tactics*. LCF introduced several tacticals that various interactive proof systems have employed. Some systems, such as Isabelle [56], Nuprl [22], Coq [24], use some of the same tacticals employed in LCF. Other systems like IMPS [27] and PVS [53,54] have combinators similar to tacticals, but named differently.

**Definition 3.3.4** [THEN] Let  $t$  and  $t'$  be tactics and  $g$  a goal. If

$$t(g) = \langle [g_1, \dots, g_n], e \rangle$$

and  $t'(g_i) = \langle \bar{g}_i, e_i \rangle$ , then  $(t \text{ THEN } t')(g) = \langle \bar{g}_1 \hat{\ } \dots \hat{\ } \bar{g}_n, e' \rangle$  where  $e'$  is the partial extract constructed from composing  $e$  with  $e_1, \dots, e_n$  using the construction in Definition 3.4.3.

**Definition 3.3.5** [ORELSEL] Let  $t_0, t_1, \dots, t_n$  be tactics and  $g$  be a goal. Let  $t = t_0 \text{ ORELSEL } [t_1, \dots, t_n]$ . Then

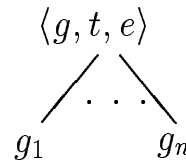
$$t(g) = \begin{cases} t_0(g) & \text{if } t_0(g) \text{ is defined} \\ t_i(g) & \text{if } t_j(g) \text{ is undefined for } 0 \leq j \leq i-1 \text{ and } t_i(g) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The tactical ORELSEL represents multiple compositions of the ORELSE tactical found in many interactive proof systems.

The process of applying a tactic to a goal is called *tactic refinement*. Using tactic refinement we develop *tactic proofs*.

**Definition 3.3.6** A tactic proof is a tree whose nodes are goals or triples  $\langle g, t, e \rangle$  where  $g$  is a goal,  $t$  is a tactic, and  $e$  is a partial extract. Tactic proofs are constructed from tactics as follows.

1. If  $g$  is a goal, then  $g$  is a tactic proof.
2. Let  $P$  be a tactic proof,  $g$  be a leaf of  $P$ , and  $t$  be a tactic. Let  $t(g) = \langle [g_1, \dots, g_n], e \rangle$ . Let  $P'$  be the tree obtained by replacing  $g$  with the tree below.



Then  $P'$  is a tactic proof.

From a tactic proof of a goal  $g$ , there exists a primitive proof of  $g$ . Furthermore, the primitive proof of  $g$  is complete only if the tactic proof of  $g$  is complete. We state this property as the following theorem.

**Theorem 3.3.7** Let  $g$  be a goal. If  $P$  is a tactic proof of  $g$ , then there exists a primitive proof  $G$  of  $g$  such that  $P$  and  $G$  have the same unrefined goals.

**Proof:** The proof proceeds by induction on the depth of  $P$ . Clearly the theorem holds for the base case, when  $P = g$ .

Suppose the depth of  $P$  is  $n$  and  $n > 1$ . Then  $P$  was obtained from a tactic proof of depth  $n - 1$ ; let  $P'$  be that proof. By the induction hypothesis, there exists a primitive proof  $G'$  of  $g$  that has the same unrefined goals as  $P'$ . Suppose that  $P'$  and  $G'$  have  $m$  unrefined goals. Moreover let the unrefined goals of  $P'$  and  $G'$  be  $g_1, \dots, g_m$  and let  $t_1, \dots, t_m$  be the tactics that refined  $g_1, \dots, g_m$  to obtain  $P$  respectively. Let  $P_1, \dots, P_m$  be the tactic proofs obtained by the tactic refinements of  $g_1, \dots, g_m$  by  $t_1, \dots, t_m$ , respectively. Then by the induction hypothesis, there are primitive proofs of  $G_1, \dots, G_m$  of the goals  $g_1, \dots, g_m$  in which each  $G_i$  has the same unrefined goals as  $P_i$ . Thus by replacing each  $g_i$  with  $G_i$  in  $G'$  yields a primitive proof of  $g$  that has the same unrefined subgoals as  $P$ .

□

Because of Theorem 3.3.7, users only need to find the tactic proof of a goal. Hence, a user considers tactics to be the means of inferences rather than refinement rules. Tactics can be designed to capture the intuition of a proof. Therefore, tactics allow users to develop formal proofs intuitively.

### 3.4 Extraction

Recall that each refined goal of a primitive proof is a triple of the form  $\langle g, r, e \rangle$  where  $r(g) = \langle [g_1, \dots, g_n], e \rangle$  and  $e$  is a *partial extract*. The partial extract constructs the *extract* of a goal. Extracts are represented as terms. The user assigns semantic meaning to an extract. The interpretation of the extract usually coincides with the interpretation of a goal. For example, if a goal represents a program specification, then the extract represents a program that implements the specification. On other hand, if the goal represents a conjecture, then the extract represents a proof of the conjecture. To make the definition of an extract precise, we follow [32] and use a binary relation  $\mathcal{E}$  over goals and extracts to determine the extract of a goal.

**Definition 3.4.1** Let  $s$  be a Nuprl term, and let  $g$  be a goal. Then  $s$  is the extract of  $g$  if  $\langle g, s \rangle \in \mathcal{E}$ .

We create an extract of a goal by applying its partial extract to the extracts of its subgoals. A partial extract is a partial function that maps a list of extracts to an extract. If  $\langle g, r, e \rangle$  is a node in a primitive proof and  $s_1, \dots, s_n$  are the extracts of its children then  $e([s_1, \dots, s_n])$  is the extract of  $g$ . We define this formally as follows.

**Definition 3.4.2** Let  $\langle g, r, e \rangle$  be a node in a primitive proof. If  $\langle g, r, e \rangle$  is a leaf then the extract of  $g$  is  $e([])$ . Otherwise, if  $\langle g, r, e \rangle$  is an interior node and the extracts of the goals of its children are  $s_1, \dots, s_n$ , then  $e([s_1, \dots, s_n])$  is the extract of  $g$ .

To obtain the extract of a goal, we compose all the partial extracts in a complete proof of the goal. We define the composition of all partial extracts in a proof in

Definition 3.4.4, but first we define the function *emap*.

**Definition 3.4.3** Let  $e, e_1, \dots, e_n$  be partial extracts and let  $k_1, \dots, k_n$  be non-negative integers. Then

$$\text{emap}(e, [\langle e_1, k_1 \rangle, \dots, \langle e_n, k_n \rangle])(s) = e([e_1(s_1), \dots, e_n(s_n)])$$

where  $s = s_1 \hat{\ } \dots \hat{\ } s_n$  and the length of each  $s_i$  is  $k_i$ .

The partial extract obtained by composing the partial extracts in the subtree rooted at  $q$  in the primitive proof  $G$  is represented as  $\text{ext}_G^q$ .

**Definition 3.4.4** Let  $G$  be a primitive proof with  $n \geq 0$  unrefined goals, and let  $q$  be a node in  $G$ . Let  $\text{ext}_G^q$  be the function from a list of extracts to extracts such that the following holds.

If  $q$  is an unrefined goal of  $G$ , then

$$\text{ext}_G^q = \lambda s. \text{head}(s)$$

where  $\text{head}([s_1, \dots, s_m]) = s_1$  for  $m \geq 1$ . If  $q = \langle g, r, e \rangle$  with children  $q_1, \dots, q_m$ ,  $m \geq 1$ , then

$$\text{ext}_G^q = \text{emap}(e, [\langle \text{ext}_G^{q_1}, k_1 \rangle, \dots, \langle \text{ext}_G^{q_n}, k_n \rangle])$$

where  $k_i$  is the number of unrefined goals in the subtree rooted at  $q_i$ . If  $m = 0$ , then  $\text{ext}_G^q = e$ .

If  $q$  is the root of  $G$  we write  $\text{ext}_G^q$  as  $\text{ext}_G$ . If  $G$  is incomplete, then  $\text{ext}_G$  represents the partial extract. Furthermore, if  $s_1, \dots, s_n$  are the extracts of the unrefined goals of  $G$ , then  $\text{ext}_G([s_1, \dots, s_n])$  is the extract of the goal at the root of  $G$ . We state this as the following theorem.

**Theorem 3.4.5** Let  $G$  be a primitive refinement proof, and let  $g$  be the goal in the root of  $G$ . Let  $s_1, \dots, s_n$  be the extracts of the unrefined goals of  $G$ . Then the extract of  $g$  is  $ext_G([s_1, \dots, s_n])$ .

**Proof:** The proof proceeds by induction on the depth of  $G$ . The base case is trivial.

For in the inductive case, suppose that the root of  $G$  is  $\langle g, r, e \rangle$ . Let  $G_1, \dots, G_m$  be the subtrees whose roots are the children of  $\langle g, r, e \rangle$ . Let  $\bar{e}_i$  be the list of extracts of the unrefined goals of  $G_i$  for  $i = 1, \dots, m$ . Then by the induction hypothesis  $ext_{G_i}(\bar{e}_i)$  is the extract of the goal in the root of  $G_i$ . Let  $s_i$  denote the extract of the goal in the root of  $G_i$ . Let  $s = \bar{e}_1 \hat{\ } \dots \hat{\ } \bar{e}_m$ . Then by Definition 3.4.4

$$\begin{aligned} ext_G(s) &= emap(e, [\langle ext_{G_1}, |\bar{e}_1| \rangle, \dots, \langle ext_{G_m}, |\bar{e}_m| \rangle])(s) \\ &= e([\langle ext_{G_1}, |\bar{e}_1| \rangle, \dots, \langle ext_{G_m}, |\bar{e}_m| \rangle])(s) \\ &= e([s_1, \dots, s_m]) \end{aligned}$$

It follows from Definition 3.4.2 that  $ext_G(s)$  is the extract of  $g$ .

□

**Corollary 3.4.6** If  $G$  is a complete primitive proof and  $g$  is the goal in the root of  $G$ . Then  $ext_G[]$  is extract of  $g$ .

### 3.5 Model of Evaluation of Tactic Refinement

In this section, we give a formal model of evaluation of tactic refinement. The model is used to define *concurrent refinement*. In addition, the model aids the

reader in understanding the implementation of tactic refinement (see Chapter 5).

Intuitively, primitive refinement is performed by a *primitive refinement machine*. The primitive refinement machine is an abstract state machine whose state is a primitive proof. The state is altered by performing primitive refinement on a goal in the primitive proof. The primitive refinement is performed with respect to a primitive refinement rule and a tree address. The primitive refinement rule and tree address represent an instruction of the primitive refinement machine. Recall from Section 3.1 that a tree address references a particular node in a tree. The tree address indicates the node in the state of the primitive refinement machine containing the goal to refine. A tactic represents a sequence of primitive refinements rules paired with tree addresses. When we perform tactic refinement of a goal  $g$  with a tactic  $t$ , we compile  $t$  into a sequence of instructions, *i.e.* a sequence of primitive refinement rules paired with tree addresses. Then we execute the instructions using  $g$  as the initial state of the primitive refinement machine. After executing all the instructions, a tactic proof is constructed from the state of the primitive refinement machine. In our model we do not explicitly define a primitive refinement machine. Instead, we define *primitive refinement traces* and describe how a primitive refinement trace is evaluated with respect to a primitive proof.

**Definition 3.5.1** Let  $A$  be the type of tree addresses (see Section 3.1 for details on tree addresses) and let  $R$  be a type containing primitive refinement rules. Then a primitive refinement trace is a list of type  $A \times R$ .

Intuitively, the primitive refinement machine uses a primitive refinement trace to direct it to the various goals in a proof to refine. Below, we define evaluating a

primitive refinement trace with respect to a primitive proof.

**Definition 3.5.2** Let  $G$  be a primitive refinement proof and let  $T = ||t(g)||_u$ . Then the value of  $\langle G, T \rangle$  is the primitive proof obtained as follows.

1. If  $T = []$  then the value of  $\langle G, T \rangle$  is  $G$ .
2. If  $T = \langle u, r \rangle \hat{\ } T'$  and  $u \in \mathcal{A}(G)$ , *i.e.* there is a node in  $G$  at the address  $u$ , then the value of  $\langle G, T \rangle$  is the value of  $\langle G[u \leftarrow G'], T' \rangle$ <sup>1</sup>:  $G'$  is the primitive proof obtained by refining the goal in  $G$  at address  $u$  with  $r$ .
3. If  $T = \langle u, t \rangle \hat{\ } T'$  and  $u \notin \mathcal{A}(G)$  then  $\langle G, T \rangle$  has no value.

We use  $\langle G, T \rangle \Downarrow G'$  to denote that  $G'$  is the value of  $\langle G, T \rangle$  and  $\langle G, T \rangle \Downarrow \perp$  to denote that  $\langle G, T \rangle$  has no value. The second part of Definition 3.5.2 permits refining goals previously refined. In other words, if  $u$  refers to a node  $\langle g', r', e \rangle$  then we replace the interior subtree rooted at  $\langle g', r', e \rangle$  with the tree obtained by refining  $g'$  with  $r$ .

We create primitive refinement traces from tactics. The primitive trace represents the primitive refinements performed when evaluating the tactic on a goal.

**Definition 3.5.3** Given a simple tactic  $t$  associated with type  $A$  containing pairs of goals and primitive proofs and a goal  $g$ , the primitive refinement trace of  $t$  applied to  $g$  at address  $u$ , denoted  $||t(g)||_u$ , is the list of pairs of primitive refinements and tree addresses,  $[\langle r_1, u \hat{\ } u_1 \rangle, \dots, \langle r_n, u \hat{\ } u_n \rangle]$  such that each  $r_i$  is contained in the node at address  $u_i$  in the primitive proof  $G$ , where  $\langle g, G \rangle \in A$ . If  $t$  is undefined on  $g$ , then

$$||t(g)||_u = []$$

---

<sup>1</sup>Recall from Section 3.1 that  $G[u \leftarrow G']$  represents the tree obtained by replacing the subtree rooted at the node at address  $u$  in  $G$  with  $G'$ .

When  $u = []$ , we abbreviate  $\|t(g)\|_u$  as  $\|t(g)\|$ .

We define the primitive refinement trace of composite tactics as follows.

**Definition 3.5.4** Let  $g$  be a goal,  $t$  be a composite tactic, and  $u$  be a tree address.

1. Suppose  $t = (t' \text{ THEN } t'')$ . If  $\|t'(g)\|_u \Downarrow G$ , then let  $g_1, \dots, g_n$  be the unrefined goals of  $G$ . Furthermore, let  $u_1, \dots, u_n$  be the tree addresses of  $g_1, \dots, g_n$  in  $G$  respectively.

Then

$$\|t(g)\|_u = \|t'(g)\|_u \hat{\ } \|t''(g_1)\|_{u \hat{\ } u_1} \hat{\ } \dots \hat{\ } \|t''(g_n)\|_{u \hat{\ } u_n}$$

If  $\|t'(g)\|_u \Downarrow \perp$ , then  $\|t(g)\|_u = []$

2. Suppose  $t = (t' \text{ ORELSE } t'')$ . Then

$$\|t(g)\|_u = \begin{cases} \|t'(g)\|_u & \text{if } t' \text{ is defined on } g \\ \|t''(g)\|_u & \text{otherwise} \end{cases}$$

As an example, we construct the primitive refinement trace for the tactic  $t$  on the goal  $g$ .

$$t = \text{AndTac THEN } t' \tag{3.3}$$

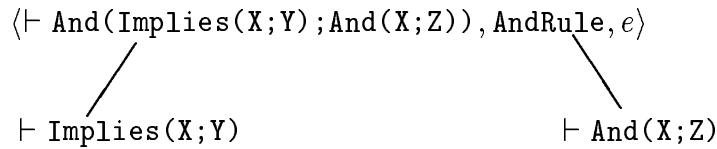
$$t' = \text{AndTac ORELSE ImpliesTac} \tag{3.4}$$

$$g = \vdash \text{And}(\text{Implies}(X;Y); \text{And}(X;Z)) \tag{3.5}$$

**AndTac** is the tactic that performs one primitive refinement using primitive refinement rule **AndRule**. **AndRule** is the refinement rule for and-introduction in the sequent calculus. **AndRule** takes a goal of the form  $H \vdash \text{And}(A;B)$  and produces subgoals  $H \vdash A$  and  $H \vdash B$ . **ImpliesTac** is the tactic that performs one refinement

using the refinement rule `ImpliesRule`. `ImpliesRule` is the refinement rule for implies-introduction in the sequent calculus. `ImpliesRule` takes a goal of the form  $H \vdash \text{Implies}(A;B)$  and produces the subgoal  $H, :A \vdash B$ . Recall that an assumption in a goal is a pair containing a variable and term (see Section 3.3.1). The assumption  $:A$  indicates that the null variable is used as the variable in the assumption and  $A$  is the term.

In order to construct  $\|t(g)\|$ , we first evaluate  $\|\text{AndTac}(g)\|$  with respect to  $g$ . That is  $\|t'(g)\| \Downarrow G'$  where  $G'$  is the tree below.



Let

$$\begin{aligned} g_1 &= \vdash \text{Implies}(X;Y) \text{ and} \\ g_2 &= \vdash \text{And}(X;Z). \end{aligned}$$

The tree addresses of  $g_1$  and  $g_2$  in  $G'$  are 1 and 2 respectively. Thus

$$\begin{aligned} \|t'(g_1)\|_1 &= \|\text{ImpliesTac}(g_1)\|_1 = \langle 1, \text{ImpliesRule} \rangle \text{ and} \\ \|t'(g_2)\|_2 &= \|\text{AndTac}(g_2)\|_2 = \langle 2, \text{AndRule} \rangle. \end{aligned}$$

Hence

$$\|t(g)\| = [\langle [], \text{AndRule} \rangle, \langle 1, \text{ImpliesRule} \rangle, \langle 2, \text{AndRule} \rangle].$$

Note that we can evaluate  $\|t(g)\|$  on goals other than  $g$ . For example,  $\|t(g)\|$  evaluates successfully to a value with respect to the goal.

$$\vdash \text{And}(\text{Implies}(\text{And}(X;Z); Y); \text{And}(X;Z)).$$

Tactic refinement relies on evaluating a primitive refinement trace of a tactic with respect to a goal. The value of the trace is used to construct a *tactic proof*.

**Definition 3.5.5** Let  $t$  be a tactic and  $g$  be a goal. Suppose the value of  $\langle g, ||t(g)|| \rangle$  is  $G$ . Let  $P$  be the tactic proof of depth two with root  $\langle g, t, ext_G \rangle$  and whose children are the unrefined goals of  $G$ . Then  $P$  is the tactic proof obtained by the tactic refinement of  $g$  by  $t$ , denoted  $g \stackrel{t}{\Rightarrow} P$ .

### 3.5.1 The Evaluation of Tactic Refinement Using Concurrency

We can modify Definition 3.5.2 to permit the evaluation of tactic refinement using concurrency. In other words, we allow the possibility of evaluating more than one primitive refinement simultaneously. We add two instructions to support evaluating multiple primitive refinement traces using And-parallelism and Or-parallelism. With And-parallelism, we refine several goals in different branches of a proof simultaneously. With Or-parallelism, we explore several alternative proofs of a goal simultaneously.

We begin our modification by redefining primitive refinement traces to be lists of members of data-type  $\mathcal{I}$ . The type constructors of  $\mathcal{I}$  are given below.

$$\text{Seq} : \mathcal{A} \times \mathcal{R} \longrightarrow \mathcal{I}$$

$$\text{And} : (\mathcal{A} \times \mathcal{I}^*)^* \longrightarrow \mathcal{I}$$

$$\text{Or} : \mathcal{A} \times \mathcal{I}^* \longrightarrow \mathcal{I}.$$

$\mathcal{A}$  is the type of tree addresses, *i.e.* the type of lists of positive integers (see Sec-

tion 3.1), and  $\mathcal{R}$  is the type of primitive refinement rules.

Members of  $\mathcal{I}$  of the form  $\text{Seq}(\langle u, t \rangle)$  are the instructions of the primitive refinement traces defined in the previous section. Members of  $\mathcal{I}$  of the form

$$\text{And}([\langle u_1, L_1 \rangle, \dots, \langle u_n, L_n \rangle])$$

specify And-parallelism, and members of  $\mathcal{I}$  of the form

$$\text{Or}(\langle u, [L_1, \dots, L_n] \rangle)$$

specify Or-parallelism. A refiner can perform concurrent refinement by evaluating tactic execution sequences that contain concurrent instructions. In the following definition, we formally define evaluation of primitive refinement traces created using members of  $\mathcal{I}$ .

**Definition 3.5.6** Let  $G$  be a primitive proof and let  $L$  be a list of members of  $\mathcal{I}$ . Then the value of  $\langle G, L \rangle$  is the primitive refinement proof obtained as follows.

1. If  $L = []$  then  $\langle G, L \rangle \Downarrow G$ .
2. If  $L = \text{Seq}(\langle u, r \rangle) \hat{\ } L'$  and  $u \in \mathcal{A}(G)$  then let  $g$  be the goal at address  $u$  in  $G$ . Then  $\langle G, L \rangle \Downarrow \langle G[u \leftarrow G'], L' \rangle$ , where  $G'$  is the primitive proof obtained by refining  $g$  with  $r$ .
3. If  $L = \text{And}(\langle u_1, L_1 \rangle, \dots, \langle u_n, L_n \rangle) \hat{\ } L'$ ,  $u_1, \dots, u_n \in \mathcal{A}(G)$ , and  $u_i \not\# u_j$  for  $i \neq j$ , then let  $g_i$  be the goal contained in the node at address  $u_i$  in  $G$  for  $i = 1, \dots, n$ . If  $\langle g_1, L_1 \rangle \Downarrow G_1, \dots, \langle g_n, L_n \rangle \Downarrow G_n$  then

$$\langle G, L \rangle \Downarrow \langle G[u_1 \leftarrow G_1, \dots, u_n \leftarrow G_n], L' \rangle.$$

4. If  $L = \text{Or}(\langle u, [L_1, \dots, L_n] \rangle) \hat{\sim} L'$ ,  $u \in \mathcal{A}(G)$ , then let  $g$  be the goal at node  $u$  in  $G$ . Let  $\langle G(u), L_1 \rangle \Downarrow V_1, \dots, \langle G(u), L_n \rangle \Downarrow V_n$  then

$$\langle G, L \rangle \Downarrow \langle G[u \leftarrow V_i], L' \rangle$$

where  $V_1 = \dots = V_{i-1} = \perp$  and  $V_i \neq \perp$  or  $i = n$ .

We modify Definition 3.5.2 to create primitive refinement traces as lists of  $\mathcal{I}$  by replacing all occurrences of  $\langle u, t \rangle$  with  $\text{Seq}(\langle u, t \rangle)$ . In addition, we define two new tacticals, PTHEN and PORELSEL, that yield primitive refinement traces using the And and Or instructions. PTHEN is the concurrent version of THEN, and PORELSEL is the concurrent version of ORELSEL. We omit the formal definition of PTHEN and PORELSEL because they are the same as the definitions of THEN (Definition 3.3.4) and ORELSEL (Definition 3.3.5).

We extend Definition 3.5.2 to support evaluation of primitive refinement traces created with lists of members of  $\mathcal{I}$  as follows.

**Definition 3.5.7** [Extension to Definition 3.5.2] Let  $t$  be a tactic and  $g$  be a goal.

1. If  $t = t'$  PTHEN  $t''$  then let  $t'(g) = \langle [g_1, \dots, g_n], e \rangle$ . If  $\langle g, ||t'(g)|| \rangle \Downarrow G$  then let  $g_1, \dots, g_n$  be the unrefined goals of  $G$  with addresses  $u_1, \dots, u_n$  respectively. Then

$$||t(g)||_u = ||t'(g)||_u \hat{\sim} [\text{And}(\langle [u_1, ||t''(g_1)|| \rangle, \dots, \langle u_n, ||t''(g_n)|| \rangle \rangle)].$$

2. If  $t = t_0$  PORELSEL  $[t_1, \dots, t_n]$  then

$$||t(g)||_u = [\text{Or}(\langle u, [||t_0(g)||_u, ||t_1(g)||, \dots, ||t_n(g)||] \rangle)].$$

# Chapter 4

## Algorithms

### 4.1 Substitution

For brevity we eluded from concretely describing *capture avoidance* in our definition of second order substitution in chapter 3. Nuprl's discipline for renaming to avoid capture is sophisticated. Nuprl requires renaming to respect *variable indifference*. Variable indifference maintains the invariant that if two bindings of two sibling subterms are identical in  $t$ , and  $t'$  is  $\sigma t$  with bound variables changed to avoid capture, then they are identical in  $t'$ . For example, maintaining variable indifference causes the result of

$$[z/x]\mathbf{holy}(x.z, x.x)$$

to be

$$\mathbf{holy}(x1.x, x1.x1)$$

instead of

$$\mathbf{holy}(x1.x, x.x).$$

The Nuprl designers chose to support variable indifference to prevent from ruining the intuitive meaning of a term implied by the user. Although a term with different names for bindings of different sibling subterms are alpha equivalent, the user may have an intuitive meaning prescribed to a term based on the names of the bindings. Programmers similarly use names of variables in programs to describe the purpose of the variable. Thus, the intuitive meaning may change by having two bindings differ that were originally the same. To further support this concept, the Nuprl designers required another feature of renaming, which we demonstrate in the following example. Substituting the second order variable instance  $f[y]$  in the term

$$A(y.f[y]) \tag{4.1}$$

with the bound term  $x.B(y.C(x; y))$  results in the substitution of  $y$  for  $x$  in

$$B(y.C(x; y)). \tag{4.2}$$

That is

$$[x.B(y.C(x; y))/f] A(y.f[y]) = A(y.([y/x]B(y.C(x; y)))). \tag{4.3}$$

Carrying out the substitution on the right hand side of (4.3) causes  $y$  to be captured. To avoid the capture, normally, the bound occurrence of  $y$  in (4.2) would be renamed. With Nuprl substitution, however, the bound occurrence of  $y$  in (4.1) is renamed to avoid capture. Thus,

$$\begin{aligned} [x.B(y.C(x; y))/f] A(y.f[y]) &= A(y.([y/x]B(y.C(x; y)))) \\ &= A(y1.(B(y.C(y1; y)))) \end{aligned}$$

compared to

$$A(y.(B(y1.C(y; y1))))).$$

Nuprl substitution also allows the user to *explicitly rename* a bound variable. Without capture avoidance, explicit renaming may cause a free variable to become bound. For example, we explicitly renaming  $u$  to  $x$  in (4.4) without capture avoidance.

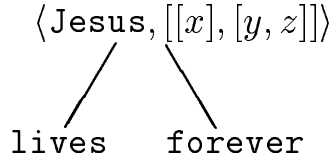
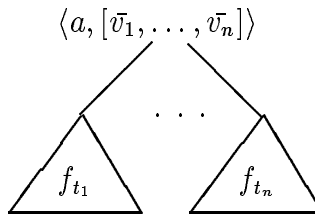
$$A(u.B(x; u)) \tag{4.4}$$

The resulting term,  $A(x.B(x; x))$ , no longer has any free occurrences of  $x$ . Nuprl substitution avoids capture when explicitly renaming a bound variable. Thus, the result of explicitly renaming  $u$  to  $x$  in (4.4) is  $A(x1.B(x; x1))$ . Nuprl substitution allows bound variables to be explicitly renamed while performing second order substitution. Users generally do not perform explicit renaming directly. Explicit renaming is primarily performed by the term abstraction mechanism.

To develop an algorithm for substitution with capture avoidance, we represent Nuprl terms as trees (See Section 3.1 regarding our definition of trees). Recall from Section 3.2.2 that  $\mathcal{T}_\nu$  is the type representing Nuprl terms. Given  $t \in \mathcal{T}_\nu$ , we interpret  $t$  as the tree  $f_t$  whose nodes are either operators or an operator paired with a list of lists of names of bindings. For example, if

$$t = \text{Jesus}(x.\text{lives}(); y, z.\text{forever}())$$

then  $f_t$  is the tree given in Figure 4.1. In general, we represent a term  $t$  as a tree  $f_t$  as follows. If  $t = a()$ , then  $f_t$  is the tree with  $a$  as its only node. If  $t = a(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$  then  $f_t$  is the tree below.

Figure 4.1: Tree Representation of  $t$ 

We develop an algorithm for Nuprl substitution that treats terms as trees. Thus, for the remainder of this section, we treat the term  $t$  as the tree  $f_t$ . Recall that we use tree addresses to refer to a specific node in a tree. Because we interpret terms as trees, trees addresses refer to a specific subterm. In other words, for a term  $t$  and address  $u$ ,  $t/u$  is the subterm of  $t$  at address  $u$ . Recall from Section 3.1 that  $t/u$  is the subtree of  $t$  rooted at the node at address  $u$ . In addition to subterms, we reference bindings of a subterm using addresses. We use tree addresses as part of *binding addresses*.

**Definition 4.1.1** A binding address of a term  $t$  is a triple  $\langle u, m, p \rangle$  where  $u$  is a tree address of  $t$ , and  $m$  and  $p$  are positive integers. The integer  $m$  refers to the  $m$ -th list,  $\bar{v}_m$ , in the node  $\langle a, [\bar{v}_1, \dots, \bar{v}_n] \rangle$  at address  $u$  in  $t$  and  $p$  refers to the  $p$ -th element of  $\bar{v}_m$ .

The binding address  $\langle [], 2, 1 \rangle$  for the term in Figure 4.1 refers to the binding  $y$ . We let  $\mathcal{B}$  denote the type of binding addressees and for a term  $t$ ,  $\mathcal{B}(t)$  is the type

containing binding addresses for bindings in  $t$ .

We extend  $\preceq$  to include binding occurrence addresses. That is, for  $\langle u, m, p \rangle \in \mathcal{B}$  and  $\langle u', m', p' \rangle \in \mathcal{B}$ ,  $\langle u, m, p \rangle \preceq \langle u', m', p' \rangle$  if and only if each of the following criteria hold.

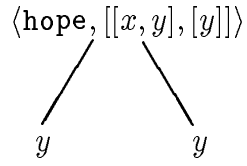
1. If  $u \neq u'$  then  $u \preceq u'$ .
2. If  $u = u'$  then  $m < m'$  or  $m = m'$  and  $p \leq p'$ .

If  $\langle u, m, p \rangle \preceq \langle u', m', p' \rangle$ , then the binding at  $\langle u', m', p' \rangle$  is in the *scope* of the binding at  $\langle u, m, p \rangle$ . The binding with address  $\langle v, m, p \rangle \in \mathcal{B}(t)$  binds the subterm  $t/v \hat{=} m$  of  $t$ . Thus, a subterm  $t/u$  is in the scope of the binding at  $\langle v, m, p \rangle \in \mathcal{B}(t)$  if  $v \hat{=} m \preceq u$ .

### 4.1.1 Avoiding Capture

We base our algorithm on the reference version of the second order substitution algorithm implemented in Nuprl 4 created by Stuart Allen, and the unpublished notes of Rich Eaton regarding Nuprl substitution. The algorithm proceeds to rename all variables, including free variables, in the *matrix* and replacements to their *binding index*: The matrix is the term being substituted into; a binding index is a unique name of a variable. After performing the substitution, the binding indexes are converted to a name closest to the original name without causing capture. Below we describe how to create a binding index and give the procedure for changing the binding index into a name to avoid capture.

A *binding collection* identifies all of the occurrences of a common binding for all the sibling subterms of a term. The binding collection, for example, of  $y$  in the term below is the list of binding addresses  $[\langle [], 1, 2 \rangle, \langle [], 2, 1 \rangle]$ .



**Definition 4.1.2** A *binding collection* is a list of binding addresses

$$[\langle u, m_1, p_1 \rangle, \dots, \langle u, m_k, p_k \rangle]$$

such that for any  $i$  and  $j$ ,  $1 \leq i, j \leq k$ ,  $M(\langle u, m_i, p_i \rangle) = M(\langle u, m_j, p_j \rangle)$ .

Binding collections are used to maintain variable indifference. Renaming one binding in a binding collection requires renaming all the bindings in the binding collection to the same name. To maintain variable indifference, we assign a binding index as a unique name for an entire binding collection.

**Definition 4.1.3** Given a binding collection  $B = [b_1, \dots, b_n]$ , the *binding index* for  $B$  is the triple  $\langle b, y, k \rangle$  where  $b \in \mathcal{B}$ ,  $y$  is the name of a binding, and  $k$  is an integer such that  $1 \leq k \leq 4$ . The binding address  $b$  is the least  $b_i$  of  $b_1, \dots, b_n$  with respect to  $\preceq$ .

For the binding index  $\langle b, y, k \rangle$  of  $B$ , the name  $y$  is the *preferred name* of  $B$ . If a binding in  $B$  is a target of explicit renaming, then  $y$  is the name of the replacement; otherwise  $y$  is the name of the bindings in  $B$ . We say that a binding collection has been *selected* if it one of its members is a target of explicit renaming. The value of  $k$  is the priority of the binding index. The priority determines whether a binding collection has to be renamed to avoid capture. One is the highest priority and five is the lowest. The priorities are assigned according to Table 4.1.

Table 4.1: Priorities of Binding Indexes

Priority	Description
1	Free variable
2	Occurs in the replacement
3	Selected for explicit renaming in the matrix
4	Occurs in the matrix, but not selected for explicit renaming
5	Does not bind any variable

The binding index of a free variable  $x$  is

$$\langle \langle [], 1, 1 \rangle, x, 1 \rangle.$$

Recall that  $[]$  is the address of the root.

After we perform substitution, we convert the binding indexes to their preferred names. Capture occurs during the conversion if a binding index  $b$  occurs in the scope of a binding index  $b'$  and both have the same preferred name. To avoid capture, we rename the preferred name of the binding index that has the lowest priority. The procedure for converting binding indexes to their preferred names collects all the binding indexes using breadth-first traversal on the term produced from the substitution. Then the procedure rearranges the list using the partial order  $\triangleleft$  defined below.

**Definition 4.1.4** Given two binding indexes  $\langle b, x, k \rangle$  and  $\langle b', x', k' \rangle$ ,

$$\langle b, x, k \rangle \triangleleft \langle b', x', k' \rangle$$

if one of the following conditions holds.

1. If  $k < k'$  then  $b \preceq b'$  or  $b' \preceq b$  and  $x = x'$ .
2. If  $k = k'$  then  $b \preceq b'$  and  $x = x'$ .

The procedure uses  $\prec$  to determine when capture occurs and which preferred name to change. If for two binding indexes  $b$  and  $b'$   $b \prec b'$ , then capture occurs. Because the priority of  $b$  is at least the priority of  $b'$ , the preferred name of  $b'$  is changed. The SML function `changePreferredName` defined below changes all preferred names in a list of binding indexes to avoid capture.

```

fun changePreferredName I = let
  (*
    I is a list of binding indexes obtained by breadth--first
    traversal
  *)

  val I' = rearrange_list_using_< (I)
  fun change ((bi,xi,ki)::tl) newI =
    if i = 1
    then
      change tl (bi,xi,ki) :: newI
    else
      if ∃ (b,x,k) in newI such that (b,x,k) < (bi,xi,ki)
      then change (bi,newUnusedName(xi),ki) :: tl newI
      else
        change tl (bi,xi@1,ki) :: newI

  | change [] newI = newI
in
  change I' []
end

```

The function `newUnusedName` picks a new name for  $x_i$  that is different from all preferred names in the binding indexes in `newI` by appending a suffix to  $x_i$ .

The function `nupr1_subst`, below, is the function for substitution with capture avoidance.

```

fun nuprl_subst M σ ρ = let
  (* σ is a substitution as defined in Chapter 3
     ρ is a list of pairs (x,y) to indicate x should be
        explicitly renamed to y
  *)
  val M' = renameBoundVarsToBindingIndexes(M)
  val M'' = changePreferredNamesOfSelectedBindings (M,ρ)
  val σ' = renameBoundVarsInReplacementstoBindingIndexes(σ)
  val N = σ'M''
  val I = getBindingIndexes(N)
  val I' = changePreferredName I
in
  convertBindingIndexesToPreferredNames (N,I')
end

```

## 4.2 Equality

In this section, we specify the algorithm for deciding equality within the Nuprl type theory. The equality refinement rule of the Nuprl type theory uses this algorithm to establish equality between two terms within a specific type based on a set of equations.

We define equality as follows. First we take the hypothesis list of the goal and take all the equations from the hypothesis list. An equation in the Nuprl type theory is a term of the form

$$\text{equal}(e_1; e_2; T)$$

where  $e_1$  and  $e_2$  represent the equands of the equation and  $T$  is a type. We write the equation as

$$e_1 = e_2 \text{ in } T.$$

The equation represents  $e_1$  and  $e_2$  being equal in type  $T$ . For more on the semantics

of Nuprl equations see [6]. Once we isolate the equations from the hypothesis list, we develop a binary relation from the equations as follows.

**Definition 4.2.1** Let  $E$  be a set of Nuprl type theory equations. Then  $\leftrightarrow_E$  is the binary relation over the terms of the Nuprl type theory such that  $s \leftrightarrow_E t$  if one of the following holds.

1.  $s \neq_\alpha t$ ,  $s' = t'$  in  $T \in E$  and either  $s =_\alpha s'$ , and  $t =_\alpha t'$ , or  $s =_\alpha t'$  and  $t =_\alpha s'$
2.  $s =_\alpha t$  and  $s' = t'$  in  $T \in E$  such that  $s =_\alpha s'$  or  $s =_\alpha t'$ .

Clearly,  $\leftrightarrow_E$  is symmetric and transitive, but not reflexive because of Condition 2. The semantics of the Nuprl type theory require that  $\leftrightarrow_E$  be non-reflexive [6,21]. According to the semantics of the Nuprl type theory, if  $x = y$  in  $T$  is true, then  $x$  and  $y$  are members of type  $T$ . Therefore,  $x = x$  in  $T$  represents the typing judgment  $x$  is a member of  $T$ . If  $\leftrightarrow_E$  was reflexive, then for any  $x$  and any type  $T$ ,  $x \in T$ .

The crux of the equality algorithm is based on Definition 4.2.2 and Definition 4.2.3.

**Definition 4.2.2** Let  $E$  be a set of equations and  $s$  and  $t$  be terms of the Nuprl type theory. Then  $s \leftrightarrow_E^1 t$  if  $s \leftrightarrow_E t$ . If there exists a term  $t'$  such that  $s \leftrightarrow_E^n t'$  and  $t' \leftrightarrow_E t$ , then  $s \leftrightarrow_E^{n+1} t$ .

If  $t \leftrightarrow_E^n t'$ , then we say that  $t'$  is *reachable by  $t$  in  $n$  steps*. We write  $s \leftrightarrow_E^+ t$  when there exists an  $n$  such that  $s \leftrightarrow_E^n t$ . Given a set of equations  $E$ , a term  $s$  and a natural number  $n$ , we define  $ec_E(s, n)$  to be a set of terms such that if  $t \in ec_E(s, n)$  then  $t$  is reachable by  $s$  in at most  $n$  steps.

**Definition 4.2.3** Let  $A$  be the set of equands of  $E$ . Then for a term  $s$  in the Nuprl type theory and natural number  $n$ ,  $\text{ec}_E(s, n)$  is the set of Nuprl type theory terms defined inductively below.

$$\begin{aligned}\text{ec}_E(s, 0) &= \{s\} \\ \text{ec}_E(s, 1) &= \{z \in A \mid z \leftrightarrow_E s\} \\ \text{ec}_E(s, n+1) &= \text{ec}_E(s, n) \cup \{z \in A \mid s \leftrightarrow_E^{n+1} z\}\end{aligned}$$

When  $E$  is apparent we write  $\text{ec}(s, n)$ . Clearly, if  $t \in \text{ec}_E(s, n)$ , then  $s \leftrightarrow_E^m t$  for  $m \leq n$ . We state this as the following lemma without proof.

**Lemma 4.2.4** Let  $t$  and  $s$  be terms and let  $E$  be a set of equations. Then  $t \in \text{ec}_E(s, n)$  if and only if  $s \leftrightarrow_E^n t$ .

The following definition is used to factor a set of equations  $E$  into a set of terms.

**Definition 4.2.5** If  $E$  is a set of equations and  $A$  is a set of terms then

$$E/A = \{x \mid x = y \text{ in } T \in E \text{ or } y = x \text{ in } T \in E \text{ and } \exists z \in A. y\alpha z\}$$

Notice that  $t \in E/\text{ec}(s, n)$  implies  $s \leftrightarrow^m t$  for  $m \leq n+1$ . We state this fact as the following lemma.

**Lemma 4.2.6** Let  $E$  be a set of equations,  $s$  and  $t$  be terms, and  $n \geq 0$ . If  $t \in E/\text{ec}(s, n)$ , then  $s \leftrightarrow^m t$ , for  $m \leq n+1$ .

**Proof:** If  $t \in E/\text{ec}(s, n)$ , then according to Definition 4.2.5, there is an equation  $t = s'$  in  $T$  in  $E$  such that  $s' =_\alpha x$  for some  $x \in \text{ec}(s, n)$ . According to Lemma 4.2.4,

```

equality(s,t,E) =
  begin
    ec <- {s};
    E' <- ∅;
    while E <> E'
    do
      E' <- E;
      ec <- E/ec;
      if t alpha eq to a member of ec
      then
        return(true);
      else
        E <- E - ec;
    od;
    return(false);
  end

```

Figure 4.2: Algorithm for Detecting  $s \leftrightarrow_E^+ t$

$s \leftrightarrow^m x$ , for  $m \leq n$ . Therefore from Definition 4.2.2,  $s \leftrightarrow^{m+1} t$ .

□

The following definition defines factoring a set of equations using a set of terms.

**Definition 4.2.7**

$$E - A = \{x = y \text{ in } T \mid \forall t \in A. x \neq_\alpha t \text{ and } y \neq_\alpha t\}$$

Note that  $E - \text{ec}(s, n)$  represents the set of equands from the equations in  $E$  that are not reachable by  $s$  within  $n$  steps. This fact follows directly from Definition 4.2.7 which we state as the following lemma without proof.

**Lemma 4.2.8** Let  $E$  and  $E'$  be a set of equations and let  $E'' = E' - \text{ec}_E(s, n)$ . Let  $A$  be the set of equands of the equations in  $E''$ . Then  $\forall x \in A, s \not\leftrightarrow^n x$ .

Although  $s \not\leftrightarrow^n x$ , it is possible that  $s \leftrightarrow^m x$  for  $m > n$ .

```

NuprlEquality ( $H \vdash s = t$  in  $T$ ) =
begin
  [T] <- typeEqClas T H;
  E <- eqsOfType [T] H;
  return (equality(s,t,E));
end

```

Figure 4.3: The Nuprl Equality Algorithm

In Figure 4.2, we present the algorithm that determines whether  $s \leftrightarrow_E^+ t$  holds. The algorithm takes as arguments a set of equations  $E$  and two terms  $s$  and  $t$  and returns true if  $s \leftrightarrow_E^+ t$  and false otherwise.

We sketch a proof of the correctness of the algorithm in Figure 4.2 as follows. The algorithm returns true only when  $t$  is alpha equal to a member of  $ec$ . According to Lemma 4.2.6, during the  $i$ -th iteration of the while loop,  $ec$  is equal to  $ec(s, i)$ . Therefore, according to Lemma 4.2.4, when the algorithm returns true  $s \leftrightarrow^i t$ .

The algorithm returns false only when the while loop terminates. The while loop terminates when  $E$  is the same for two consecutive iterations. The value of  $E$  is the same for two consecutive iterations when the value of  $ec$  is the same for two consecutive iterations. The value of  $ec$  can only be the same for two consecutive iterations if  $ec$  contains no members alpha equivalent to an equand in an equation in  $E$ . By Lemma 4.2.4,  $s \not\leftrightarrow^i t$  and  $s \not\leftrightarrow^{i+1} t$  according . If  $ec$  is the same for iterations  $i$  and  $i + 1$ , then  $ec$  is the same of all iterations  $j$  for  $j \geq i$ . Hence  $s \not\leftrightarrow^+ t$ .

### 4.2.1 The Nuprl Equality Algorithm

At the beginning of Section 4.2, we mentioned that equality in the Nuprl type theory is performed with respect to types. However, the algorithm in Figure 4.2 omits types. In Figure 4.3, we give the algorithm that implements Nuprl equality with respect to types. The algorithm takes as an argument a goal  $H \vdash s = t$  in  $T$ . The algorithm determines equality using the equations in  $H$  that have types equivalent to  $T$ . Type  $T$  is equal to type  $T'$  if  $T =_{\alpha} T'$  or  $T \leftrightarrow_K^+ T'$  where  $K$  is the type equations of the hypothesis list: A type equation is an equation of the form  $A = B$  in  $U$  where  $A$  and  $B$  represent types in the Nuprl type theory and  $U$  is a type universe, a term of the form  $\text{universe}\{\alpha:1\}$ . Given a set  $K$  of type equations and an equand  $T$  of one of the equations of  $K$ , we denote  $[T]$  as the *type equivalence class of  $T$  with respect to  $K$* .

**Definition 4.2.9** Let  $T$  be a type and let  $K$  be a set of equands from a set of type equations. Then

$$[T] = \{T' \mid T =_{\alpha} T' \text{ or } T \leftrightarrow_K^+ T'\}.$$

We present the algorithm for Nuprl equality in Figure 4.3. The routine `typeEqClass` extracts the equations from the hypothesis list that contain a type component that is a member of  $[T]$ ; `eqsOfType` obtains the equations from the hypothesis list that contain the types in  $[T]$ ; and `equality` is the algorithm in Figure 4.2.

# Chapter 5

## Components of the Refiner

The refiner is a program that provides support for performing tactic refinement as defined in the mathematical description of the refiner (Chapter 3). Typically, the refiner is an interactive evaluation loop for a dialect of ML that has native support for performing tactic refinement. The SML refiner is an interactive top loop for SML and Nuprl ML. Nuprl ML code has to be provided as an abstract syntax tree encoded as an S-expression. The SML refiner evaluates a Nuprl ML abstract syntax tree by first converting it to an SML abstract syntax tree.<sup>1</sup> Then the SML refiner type checks the abstract syntax tree and compiles it into machine code. Then the SML refiner executes the machine code.

The *kernel* of the refiner implements terms and tactic refinement. The refiner provides primitive types, values, and functions in the interactive evaluation loop as an interface to the kernel. Therefore, we call the interface to the refiner *primitive ML*. The kernel consists of the term module, the rule interpreter, and the proof

---

<sup>1</sup>Karl Crary developed the translator from Nuprl ML asts to SML asts (see Section 2.2).

manager. Tables from the various components of the kernel and the portion of the top level environment that contains references make up the *refiner state*. The term module implements the Nuprl term language as defined in Section 3.2. The rule interpreter defines a syntax for terms that represent specifications for primitive refinement rules. In addition, it implements primitive refinement as defined in Section 3.3.1. The proof manager is responsible for implementing tactic refinement using the term module and the rule interpreter. In the remainder of this chapter we describe the above components of the refiner. We conclude the chapter with a description of the refiner state.

## 5.1 Term Module

The term module implements the Nuprl term language described in Chapter 3. The term module was designed to be compatible with the implementation of the Nuprl term language in the Nuprl 4 refiner constructed by Rich Eaton.

### 5.1.1 Implementing Parameters and Terms

Terms in the Nuprl term language are built up from classes of primitive objects[3]. The Nuprl term language is implicitly parameterized by these classes. The primitive objects occur in terms as *parameters*. A parameter consists of the primitive object along with a tag indicating the class the object belongs. We call the tag the *parameter type* and the object of the parameter is called the *parameter value*. A parameter value is either an object of a class or a variable symbol. A parameter whose parameter value is a variable symbol is called a *meta-variable parameter*. A meta-variable parameter may be replaced by any parameter that has the same

```

datatype parm_type =
  TOKEN
| NATURAL
| LEXPRESSION (* level expression *)
| STRING
| VARIABLE
| BOOL

datatype meta_type = Abs | Disp | MetaNil

datatype edit_type = Slot | Error | EditNil

```

Figure 5.1: Implementation of Parameter Tags

parameter type as the meta-variable parameter. There are two occasions in which meta-variable parameters are used, *term abstractions* and *display forms*. Term abstractions are analogous to term definitions or rewrite rules. Display forms are user-defined directives for pretty printing terms. The mechanism that implements display forms is not part of the refiner. In addition to a meta-variable, a parameter may carry information used when creating a term with the *term editor*. The term editor is not part of the refiner. Two additional tags other than the parameter type indicate a parameter is a meta-variable or contains editing information. All three tags are implemented as datatypes with nullary type constructors Figure 5.1.

The two constructors of `meta_type` in Figure 5.1, `Abs` and `Disp`, represent the abstraction and display form parameters, respectively. The `MetaNil` constructor indicates that a parameter is not a meta-variable. Likewise `EditNil` indicates that the parameter contains no editing information. The other two constructors, `Slot` and `Error`, indicate the kind of editing information the parameter contains. Each

of the type constructors in the definition of `parm_type` in Figure 5.1 corresponds to the six classes of primitive objects that parameterized both the Nuprl 4 refiner and the SML refiner. However only the Nuprl 4 refiner allows users to parameterize the refiner with additional classes. The six classes of primitive objects are tokens, strings, variable symbols, natural numbers, booleans, and level expressions. The Nuprl 4 refiner implements tokens and strings differently, but they are the same in the SML refiner. To remain compatibility with the Nuprl 4 refiner, the SML refiner defines distinct tags for tokens and strings. Natural numbers are represented as non-negative members of the ML type `int`, and booleans are the members of the ML type `bool`. Level expressions and variable symbols are implemented as datatypes `variable` and `level_expression`. Level expressions are the indexes for type universes [72,58,6]. (Read Appendix B for a rigorous presentation of level expressions.) The class of variable symbols are used to represent the names of variables of the Nuprl term language. Recall from Chapter 3 that a first order variable is of the form `variable{x : v}` where  $x : v$  is a parameter in which  $x$  is a variable symbol and  $v$  is the parameter type of variable symbols.

In the SML refiner, parameters are implemented as four element records depicted in the following data type.

```
datatype parameter = Parm of {edit_type : edit_type,
                             meta_type : meta_type,
                             parm_type : parm_type,
                             value : string}
```

For example, the natural number parameter `3 : natural` is represented as

```
{edit_type = EditNil, meta_type = MetaNil,
  parm_type = NATURAL, value = "3"}.
```

We store the value of a parameter as a string for compatibility with Nuprl 4. For

example, the function

```
destruct_natural_number_parameter : parameter -> int
```

returns the integer represented by a natural number parameter.

Recall from Chapter 3 that parameters are part of the operator of a term. An operator is a list of parameters in addition to an operator identifier. Thus, we implement operators as the inhabitants of the type `operator` given below.

```
type operator = string * parameter list
```

Using the definition of operators, a straight forward implementation of Nuprl terms follows directly from the mathematical definition of terms in Chapter 3.

```
datatype term =
  NuprlOp of string
  | NuprlApp of operator * sos_term list
and sos_term =
  Op of string
  | Abs of variable list * sos_term
  | App of operator * sos_term list
```

However, our implementation of terms follows that of Nuprl 4. In Nuprl 4, terms are implemented as inhabitants of the following type.

```
datatype term' = Term of operator * (variable list * term') list.
```

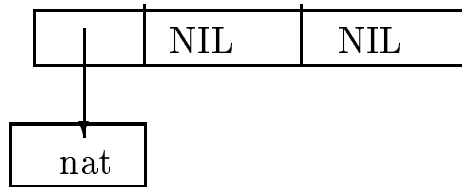
The above datatype does not explicitly distinguish variables from other terms. We distinguish between variables and other terms because variables have a different role from other terms; they may be replaced. Recall from Section 3.2 that the variables of the Nuprl term language are terms of the form `variable{v : v}()` and `variable{v : v}(t1; ... tn)`. Therefore, we implement terms using the data type in Figure 5.2. Notice that we represent the operator explicitly in this representation of terms.

```

datatype term =
  Var of variable
| SoVar of variable * term list
| Term of string * parameter list * (variable list * term) list

```

Figure 5.2: Implementation of Terms

Figure 5.3: Pointer representation of  $\text{nat}\{\}\()$ 

We can optimize the memory requirements needed to store terms by using the fact exploited in Coq [24]: Most terms contain zero, one, or two immediate subterms.<sup>2</sup> From the 41 kinds of non-canonical and canonical terms of the Nuprl type theory listed in [22], only seven have more than two immediate subterms. In our implementation a term with no immediate subterms, say  $\text{nat}\{\}\()$ , requires at most three pointers as depicted in Figure 5.3. However, we only need at most two pointers as indicated in Figure 5.4. As a result, we can implement terms as a data type with separate constructors for terms with zero, one, and two immediate subterms, and one for all other terms. This leads to the following data type for terms.

```

datatype term =
  Var of variable
| SoVar of variable * term list

```

---

<sup>2</sup>Chet Murthy is credited for discovering the memory optimization.

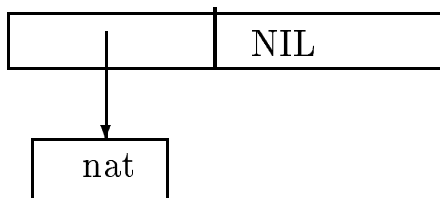


Figure 5.4: Optimize pointer representation of  $\text{nat}\{\}\()$

```

| Op0 of string * parameter list
| Op1 of string * parameter list * (variable list * term)
| Op2 of string * parameter list * (variable list * term)
      * (variable list * term)
| Opn of string * parameter list * (variable list * term) list

```

We can even make further optimizations considering very few terms have more than one parameter. We were unaware of the optimization when we implemented the SML refiner. Therefore, we use the implementation of terms in Figure 5.2.

The term module also defines various functions for operating on terms and parameters. They are listed in Appendix C with other interface functions to the refiner.

### 5.1.2 Term Evaluation

The term module implements term evaluation as defined in Section 3.2.3. Recall from Section 3.2.3 that we generalized term evaluation defined in Appendix C of [22]. Term evaluation is implemented using the term evaluator of the term module. The Nuprl 4 refiner uses evaluation rules based on the evaluation rules in [22]. The evaluation rules are hard coded into the Nuprl 4 refiner. The SML refiner, however, uses evaluation rules specified by the user. Both the Nuprl 4 and

the SML term evaluator require users to specify rewrite rules. In Nuprl 4, rewrite rules are specified as term abstractions.

We give the evaluation rule for beta reduction below.

$$\text{!eval\_rule}\{\}(ap\{\}(f; a); f; \lambda x.b \rightarrow b[a/x]) \quad (5.1)$$

Note that (5.1) is a term. The first subterm of (5.1) is the pattern that represents the terms that the rule may be applied. Both,  $f$  and  $a$  are first order variables. That is,  $f$  and  $a$  are the terms  $\text{variable}\{f : v\}()$  and  $\text{variable}\{a : v\}()$ , respectively. As in Chapter 3, we only write the variable symbol of a first order variable. The second subterm of (5.1),  $f$ , represents the *principle argument* of the evaluation rule. Although (5.1) has one principle argument, in general, an evaluation rule can have zero or more principle arguments. The last subterm of (5.1) is a reduction rule. We have written the pretty printed version of the reduction rule for clarity. In general, the last subterm may be a list of reduction rules and *meta-evaluation functions*. Meta-evaluation functions represent operations defined for the primitive objects that parameterize the refiner. For example, (5.2) below contains a meta-evaluation function.

$$\text{!eval\_rule}\{\}(add\{\}(m; n); \langle m, n \rangle; \text{!fun}\{\text{addNats : t}\}(m; n)) \quad (5.2)$$

The function `addNats` is defined as follows.

```
fun addNats x y = makeNatTerm(destNatTerm(x) + destNatTerm(y))
```

The role of `addNats` is to add the integers contained in the parameters of a natural number term and create a new term from the result.

To evaluate a term based on an evaluation rule, we first match the term against the pattern of the evaluation rule to obtain the principle arguments. For example, if

we were using (5.2) to evaluate  $\text{add}\{\}(3; 4)$  then the principle arguments would be 3 and 4. Note that we are abbreviating the natural number terms  $\text{natural}\{3 : n\}()$  and  $\text{natural}\{4 : n\}()$  as 3 and 4, respectively. Afterward, the principle arguments are evaluated. The evaluations of 3 and 4 are obviously 3 and 4, respectively. Then, the evaluated principle arguments are used to match the redexes of the reduction rules or passed as arguments to a meta-evaluation functions. The evaluation completes with the performance of the reduction indicated by a reduction rule or the result of a meta-evaluation function. If there is more than one reduction rule or meta-evaluation function, then the term evaluator tries each reduction rule and meta-evaluation function until one succeeds. In other words, the first reduction rule encountered in which the principle arguments match the redex or the first meta-evaluation function that returns successfully is used to evaluate the term. In our example, the terms 3 and 4 are passed to `addNats` yielding the term 7 as the result of the evaluation.

As indicated in Section 3.2.3, term evaluation uses rewrite rules and evaluation rules. For the term evaluator to evaluate a term, the term must be tagged. A tag is a wrapper for a term to indicate the number of steps of evaluation should be involved. For instance,  $\text{tag}\{n : \text{natural}\}(t)$  is the term  $t$  tagged to undergo  $n$  steps of evaluation. Tag terms can occur as subterms as non-tag terms. The term evaluator will evaluate the tagged subterms and replace them with the terms produced from evaluation.

## 5.2 Rule Interpreter

The rule interpreter performs primitive refinement using primitive refinement rules supplied by the user. Rich Eaton created the rule interpreter for the Nuprl 4 refiner to proven hard coding refinement rules into the refiner as in Nuprl 3. We implemented the rule interpreter in the SML refiner to have the same functionality as the Nuprl 4 rule interpreter.

We implement the mathematical description of primitive refinement rules using two objects, rule specifications and primitive rules. The user specifies primitive refinement rules as rules specifications. From a rule specification, the rule interpreter creates a primitive rule.<sup>3</sup> Primitive rules are ML objects that can be employed in tactics to perform refinement.

```

type ref_rule_spec =
  {goal : assumption_scheme list * term_scheme,
   extract : term_scheme,
   rule_name : string,
   rule_args : rule_arg_scheme list,
   side_conds : side_cond_scheme list,
   subgoals : subgoal_scheme list}

```

Figure 5.5: Type of a Primitive Refinement Rule Specifications

A rule specification is an object of type `ref_rule_spec` in Figure 5.5: The field `goal` indicates the class of goals in the rule refines; the field `extract` specifies the partial extract; the field `rule_args` specifies the kinds of objects that may be passed as arguments to the rule; the field `rule_name` indicates the name of the rule; the

---

<sup>3</sup>Considering that the rule interpreter performs a one time conversion of a rule specification into a primitive rule, a better name for it should be a rule compiler.

field `side_conds` specifies the side conditions of the rule; and the field `subgoals` are patterns for constructing subgoals. The types in the definition of `ref_rule_spec` in Figure 5.5 represent the schemes used for matching and instantiation during a refinement rule application. The *term scheme* is the most significant scheme. Term schemes specify patterns for terms. All of the other schemes contain term schemes.

Term schemes are a compact representation of terms, whose subterms have at most one parameter, for fast matching and substitution. Term schemes are expressive enough to represent the object language of most logics. In fact, all the canonical and non-canonical terms of the Nuprl type theory are expressible as term schemes. The `OpId` scheme encodes terms which only consist of an operator identifier. In other words, terms of the form  $s\{\}\()$  are encoded as `OpId s`. Terms of the form  $a\{\}(t_1; \dots t_n)$  are encoded using the `App` constructor. For example, `App(f, [s1, ..., sn])` represents the term  $f\{\}(s_1; \dots; s_n)$ . The term schemes of the form `Abs([v1, ..., vn], t)` encode bound terms with bindings  $v_1, \dots, v_n$  and body  $t$ . Term schemes of the form `TermHole x` encode first order variables. They are replaceable by terms. The `PrimObj` constructor is used to encode terms of the form  $a\{p : t\}$  such as `natural\{2 : n\}()`. Term schemes of the form `PrimObjHole(h, t)` encode meta-variable parameters. These schemes can be replaced by any parameter with parameter type  $t$ .

The inhabitants of `subst_scheme` are called *substitution schemes*.

```
datatype subst_scheme =
  SubstPat of term_scheme * (string * term_scheme) list
  | SubstHole of term_scheme * string
```

A substitution scheme is used to specify substitution within a primitive refinement

rule. To illustrate the role of a substitution scheme consider the beta-reduction rule of the lambda calculus.

$$(\lambda x.b)a \rightarrow b[a/x]$$

The right hand side of the rule is a substitution scheme. The expression  $b[a/x]$  indicates that the free variable substitution should be performed using the instantiations of  $b$ ,  $a$ , and  $x$ . In other words,  $b[a/x]$  implies that free occurrences of the variable obtained from instantiating  $x$  should be substituted with the term resulting from instantiating  $a$  in the term resulting from instantiating  $b$ .

The `SubstHole` substitution scheme allows a substitution list to be passed as an argument to the rule. The substitution list will be used to perform parameter substitution. Therefore, the targets of the substitution list must be meta-variable parameters and the replacements must be parameters.

Matching a term against a term scheme occurs within two phases. The first phase obtains a scheme substitution for all the scheme variables not contained within substitution schemes. Also during the first phase, the substitution schemes are put on a stack with their corresponding instances. In the second phase, the stacked substitution schemes are instantiated using the scheme substitution from the first phase. The instantiated substitution schemes are then tested for alpha equality with their corresponding instances.

Term schemes occur within *assumption schemes*. Assumption schemes match assumption lists within a goal. Matching an assumption list against a list of patterns is also performed in traditional proofs systems such as the sequent calculus.

For example, consider the left and-introduction rule of the sequent calculus.

$$\frac{\Gamma, \phi, \psi, \Delta \vdash \theta}{\Gamma, \phi \wedge \psi, \Delta \vdash \theta}$$

$\Gamma, \phi, \psi, \Delta$  correspond to a list of assumption schemes.  $\Gamma$  and  $\Delta$  are assumption schemes that match lists of assumptions, and  $\phi$  and  $\psi$  are assumption schemes that match a single assumption. However, matching an assumption list against the above patterns is nondeterministic. For example, using  $\Gamma, \phi, \psi, \Delta$  as a pattern to match the assumptions of the sequent  $A, B, C, D \vdash E$ , yields one of the following scheme substitutions.

$$[\langle \Gamma, [A] \rangle, \langle \phi, B \rangle, \langle \psi, C \rangle, \langle \Delta, [D] \rangle]$$

$$[\langle \Gamma, [] \rangle, \langle \phi, A \rangle, \langle \psi, B \rangle, \langle \Delta, [C, D] \rangle]$$

To make matching against a list of assumption schemes deterministic, we partition the assumption lists into segments. Then each segment is matched individually to the appropriate assumption scheme. We partition an assumption list using the function specified below.

$$\text{split}(i, l) = \begin{cases} ([], l) & \text{if } i = 0 \\ ([d_1, \dots, d_i], [d_{i+1}, \dots, d_m]) & \text{if } 0 < i < m \\ (l, []) & \text{if } i \geq m \end{cases}$$

Given a list of natural numbers  $I = [i_1, \dots, i_n]$  in increasing order and a list of assumptions  $A = [a_0, \dots, a_m]$  where  $m \geq i_n$ , we define the *segmented assumption list* of  $A$  with respect to  $I$ ,  $\text{segment}(A, I)$ , to be  $[H_1, \dots, H_n, J_n]$  where

$$(H_k, J_k) = \text{split}(J_{k-1}, i_k - i_{k-1} - \dots - i_1)$$

for  $k = 1, \dots, n$  and  $J_0 = [a_0, \dots, a_m]$ . For example, for assumption list  $A = [V, W, X, Y, Z]$  and  $I = [1, 3]$

$$\text{segment}(A, I) = [[V], [W, X], [Y, Z]]$$

The numbers used to segment an assumption list are passed as arguments to a refinement rule. We call them *assumption indexes*.

Some refinement rules use decision procedures in addition to pattern matching. Nuprl 4 provides *let expressions* for specifying decision procedures as side conditions. Let expressions are similar to `val` expressions in SML. Intuitively, a let expression basically implements SML pattern matching within a refinement rule (see Section 2.1.5).

A side condition scheme is intuitively an expression of the form

$$\text{val}\langle \text{term schemes} \rangle = \langle \text{term schemes} \rangle \tag{5.3}$$

or

$$\text{val}\langle \text{term schemes} \rangle = \langle \text{function} \rangle. \tag{5.4}$$

The term schemes on the left hand side of (5.3) and (5.4) represent a list of patterns. The term schemes on the right hand side of (5.3) represent a list of objects that evaluate to terms: They are evaluated by instantiating them using the scheme substitution obtained from matching the goal and the rule arguments. The outcome of the instantiation is then matched against the term schemes on the left hand side. The resulting scheme substitution is combined with the substitution obtained from matching the goal and the rule arguments. The combined scheme

```

datatype refine_rule =
  RefineRule of
    {arguments : argument list -> scheme_subst
      goal : sequent -> argument list -> scheme_subst -> scheme_subst,
      sideConds : sequent -> argument list -> scheme_subst
                    -> scheme_subst,
      subgoals : scheme_subst -> sequent -> (sequent * bool) list,
      extract : term_scheme * string list,
      name : string}

```

Figure 5.6: Type of Refinement Rules

substitution is used to instantiate the subgoal schemes to produce subgoals. Let expressions of the form of (5.4) are evaluated similarly except the function is applied to the goal and the rule arguments to obtain values for the right hand side. The function is a decision procedure. The Nuprl type theory rule for equality uses the equality algorithm from Section 4.2 to determine whether two terms in the conclusion of a goal are equal. The function only returns if the terms are equal; otherwise it raises an exception. In the Nuprl 4 refiner, the functions may return subgoals. The subgoals are used as the subgoals produced by the rule. The arith rule uses a let expression whose function generates a list of goals.

### 5.2.1 Implementation of a Refinement Rule

The rule interpreter converts a refinement rule specification into a refinement rule. In the SML refiner, a refinement rule is a member of type `refine_rule`.

The refinement rule contains functions that performs various matching and instantiation as indicated in Figure 5.6. A refinement rule is applied to a list of rule arguments and a goal using the function `applyRule` in Figure 5.7. When applying

a refinement rule to a goal and a list of arguments, the rule arguments are matched first using `arguments` (see Figure 5.6). Next, the goal is matched using the function `goal`. The function `goal` combines the substitution into a single substitution. That is why the substitution resulting from matching the rule arguments is a parameter to `goal` (see Figure 5.7). In addition, `goal` takes the rule argument instances as a parameters because they may contain assumption indexes for segmenting assumptions. The substitution returned by `goal` is used. The function `goal` combines the substitutions. The substitution returned by `goal` is used to evaluate the side conditions using `sideConds`. The result of evaluating the side conditions is a substitution that we combine with the substitution generated from matching the goal. The combined substitution is used to generate subgoals by instantiating subgoal schemes using the function `subgoals`. Finally, a partial extraction is created and

```

fun applyRule rule argInstances goalInstance = let
  val RefineRule {arguments,goal,extract,
                  sideConds,subgoals,name} = rule
  val sigma' =
    goal goalInstance argInstances (arguments argInstances)
  val sigma = sideConds goalInstance argInstances sigma' @@ sigma'
  val subgoals = subgoals sigma goalInstance
in
  (Extract (#1 extract, #2 extract, sigma), subgoals)
end

```

Figure 5.7: Function for Applying Refinement Rules

returned with the subgoals. Intuitively, the partial extraction is  $(\lambda s_1, \dots, s_n. t)$  where  $s_1, \dots, s_n$  are the scheme variables of `extract` (see Figure 5.6) and  $t'$  is the term scheme produced after instantiating the term scheme of `extract` using the

same substitution used to instantiate the subgoals.

### 5.3 Proof Manager

Recall from Chapter 3 that we distinguish between inference performed using primitive refinement rules and tactics. Conducting inference using primitive refinement rules produced primitive refinement proofs. Conducting inference using tactics produced tactic proofs. Both kinds of proofs are represented as trees. A node of a primitive refinement proof is either a goal or a triple  $\langle g, r, e \rangle$ , where  $g$  is a goal,  $r$  is refinement rule, and  $e$  is a partial extract. Likewise, a node in a tactic proof is either a goal or a triple  $\langle g, t, e \rangle$ , where  $g$  is a goal,  $t$  is a tactic, and  $e$  is a partial extract. In both kinds of proofs, all interior nodes are triples. Furthermore, the goals in the children of a node  $\langle g, j, e \rangle$  can be obtained by refining  $g$  using the tactic or refinement rule  $j$ . Because primitive refinement proofs and tactic proofs closely resemble each other, we implemented them using a single data structure of type `proof`. Intuitively, the structure of members of `proof` are the same as primitive refinement proofs and tactics proofs: All nodes are either goals or triples. The second component of a triple  $\langle g, j, e \rangle$  is a member of type `rule`. Members of type `rule` are either primitive refinement rules, as defined in the previous section, or an ML expression. The function `term_to_rule` is the only constructor of `rule`.

```
val term_to_rule : term -> rule
```

The term passed to `term_to_rule` represents a command for retrieving a refinement rule from the refinement rule table or ML code for representing a tactic. The only constructor of `proof` is the function `make_proof_node`.

```
val make_proof_node: assumption list -> term -> proof
```

The function `make_proof_node` creates a goal, a proof with one node. The assumption list and term passed to `make_proof_node` represent the hypothesis and the conclusion of the goal. Users refine the goal using a rule in conduction with the function `refine`.

```
val refine: rule -> proof -> (proof list *(proof list -> proof))
```

Applying `refine` to a member of `rule` representing a primitive refinement rule or representing an ML expression produces a tactic. A tactic in Nuprl is any ML function with the following type.

```
type tactic = proof -> (proof list * (proof list -> proof)).
```

The tactic produced when applying `refine` to a primitive refinement rule performs primitive refinement using the refinement rule. For instance, suppose `r'` has type `rule` and represents the primitive refinement rule `r`. Then the declaration

```
val f = refine r'
```

makes `f` the tactic representing the refinement rule `r`. Given a proof `p`, the declaration

```
val (pfs, v) = f p
```

makes `pfs` the subgoals produced by refining the goal at the root of `p` by `r`. The object `v` is a *validation*. A validation is a function of type

```
proof list -> proof
```

that constructs the proof of a goal given proofs of its subgoals. For example, applying `v` to `pfs` produces the proof of depth two in which `g` is the root and `pfs`

are its children. The validation  $v$  succeeds to produce a proof of  $g$  given any proofs of the subgoals  $pfs$ . For example, if  $pfs'$  is the list of completed proofs of each subgoal in  $pfs$ , then the expression  $v(pfs')$  evaluates to a proof in which the root contains  $g$  and the immediate subtrees of the root are the proofs  $pfs'$ .

An object of type `rule` representing an ML expression has type `tactic`. Therefore, applying the expression to a goal produces a list of subgoals and a validation. However, the application does not perform tactic refinement. Tactics produced from applying `refine` to a rule representing an ML expression of type `tactic` perform tactic refinement. For example, let  $r$  be a rule representing the expression  $t'$  of type `tactic`. The declaration

```
val t = refine r
```

binds  $t$  to a value of type `tactic`. Given a goal  $g$ , the declarations

```
val (pfs,v) = t g
val (pfs',v') = t' g
```

both bind  $pfs$  and  $pfs'$  to a list of goals and  $v$  and  $v'$  to validations. In fact, the list of goals  $pfs$  and  $pfs'$  are equal. However, the proofs produce by  $v$  and  $v'$  are not equal. The proof produced by  $v$  is a condensed version of the proof produced by  $v'$ . For example,  $v(pfs)$  is a proof of depth two: The root contains  $g$ , and its children are the goals in  $pfs$ . The proof  $v'(pfs')$ , however, may have depth greater than two, but the root contains  $g$  and the unrefined goals are  $pfs'$ . We define the function for performing tactic refinement in Figure 5.8. In Figure 5.8,  $r$  is a rule representing an ML expression of type `tactic`;  $pf$  is a proof;  $g$  is the goal in the root of  $pf$ ;  $t'$  is the ML expression contained in  $r$ ;  $p$  is the proof constructed when applying  $t'$  to  $g$ ;  $pfs'$  are the unrefined goals of  $p$ ; and  $v$  is validation that

```

fun refineTactic r pf =
  val g = goalOfRoot pf
  val t' = tacticExpressionOfRule r
  val (pfs',v') = t' g
  val p = v'(pfs')
  val v = condensedValidation p
in
  (pfs', v)
end

```

Figure 5.8: ML code for Performing Tactic Refinement

constructs a condensed version of  $p$ .

In addition to performing tactic refinement, the proof manager maintains the *theorem table*. We consider a theorem to be a proof without any unrefined goals. The statement of a theorem is the goal at the root of the theorem. Each entry in the theorem table contains the statement, `extract`, and `term_of` term of a theorem. Recall from Chapter 3, that the extract of a proof  $P$  is the term  $ext_P$ . The `term_of` term and the `extract` are used to create a rewrite rule: The `term_of` term is the left hand side of the rewrite rule and the `extract` is the right hand side of the rewrite rule. Rewrite rules created from theorems are used in term evaluation (see Section 5.1).

## 5.4 Refiner State

The refiner state contains the ML state, the theorem table, the rewrite rule table, the evaluation rule table, the side condition table, and the refinement rule table. The ML state is the part of the top level environment of the refiner that pertains to reference cells defined by users. Reference cells are mutable values in ML that are

similar to regular variables in imperative programming languages. We defined the theorem table in Section 5.3. The side condition table is used to store functions invoked during primitive refinement as described in Section 5.2. The refinement rule table is actually maintained by the interface layer of the refiner. The refinement rule table contains the refinement rules that tactics use to perform primitive refinement. The rewrite rule table and the evaluation rule table contain rewrite rules and evaluation rules used to evaluate terms. The evaluation rule table also contains meta-evaluation functions used in the evaluation rules. The interface layer of the refiner provides functions so that tactics and other ML expressions can access the various tables in the refiner state. In addition, ML expressions can alter the refiner state via assignments to reference cells. Therefore, a tactic may have different outcomes on different values of the refiner state. In Nuprl 4, there is no mechanism to ensure that the refiner state will be the same for every invocation of a tactic. The user must ensure that a tactic is invoked within the proper refiner state to produce the desired result.

## **Part III**

# **Concurrent Refinement**

## Chapter 6

# Implementation of Concurrent Refinement

Concurrent refinement is implemented using the *MP refiner*. The MP refiner consists of a collection of refiners that share the same refiner state<sup>1</sup>. Any refiner in the collection may request that another refiner evaluate a tactic application. In Figure 6.1, for example, the refiner *R* spawns the application of the tactic *t* to the goal *g* by placing  $\langle t, g \rangle$  in the *request queue*. The refiner *P* dequeues  $\langle t, g \rangle$  from the request queue and applies *t* to *g*. Afterward, *P* places the result of the application in the *reply table*. *R* retrieves the results of the application from the reply table. In the remainder of this chapter we describe the implementation of the MP refiner and how the MP refiner supports concurrent refinement through concurrent tactics. We conclude the chapter with a discussion on the non-deterministic behavior of concurrent refinement in Section 6.3.

---

<sup>1</sup>See Section 5.4 for details regarding the refiner state

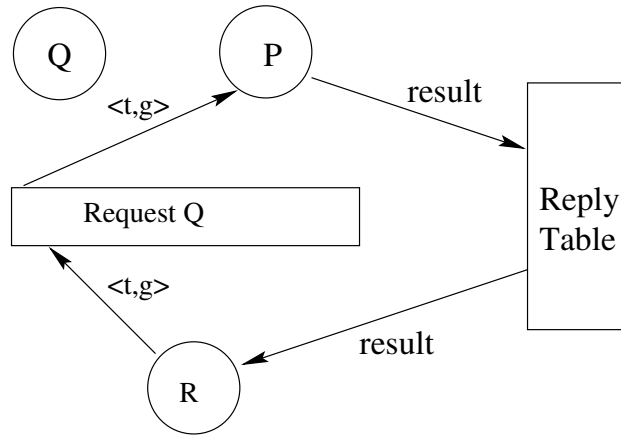


Figure 6.1: MP Refiner

## 6.1 Implementation of the MP refiner

We implemented the MP refiner in version 1.09.10 of SML/NJ with the **MP** runtime system for Solaris. Currently, the MP refiner runs on a Sun Sparc 670 shared memory multi-processor with 4 processors and is capable of running on any Sun Sparc multi-processor with at least 3 processors under Solaris 2.5.

The MP refiner consists of multiple ML threads. One thread behaves as a normal refiner: It is an interactive ML top-loop with primitive support for developing proofs using tactic refinement. The other threads are ML read-eval loops that only evaluate tactic applications. Each thread runs on a unique proc. Recall from Section 2.3.1 that a proc represents a virtual processor that executes an ML thread.

The request queue and the reply table are implemented using the structure `RefinementServer`. The signature of `RefinementServer` is given in Figure 6.2. The request queue is a queue of objects of type

```

signature REFINEMENT_SERVER:
sig
  eqtype rereq_id
  eqtype server_id
  type reply
  type server
  type request_handler
  val enqRequest : (tactic * proof) -> req_id
  val enqRequests : (tactic * proof) list -> req_id list
  val deqRequest : unit -> (req_id * (tactic * proof)) option
  val deqRequests : int -> (req_id * (tactic * proof)) list
  val insertReply : req_id * reply -> unit
  val getReply : req_id -> reply option
  val waitForReply : req_id -> request_handler -> reply
  val allReplies : req_id list -> request_handler -> reply list
  val discardReplies : req_id list -> unit
  val server : int -> ((tactic * proof)
    -> (proof list * (proof list -> proof)))
    -> server
  val spawn: server -> server_id option
  val processReply : reply -> (proof list * (proof list -> proof))
  val requestHandler : ((tactic * proof) ->
    (proof list * (proof list -> proof)))
    -> request_handler
  val serviceRequest : request_handler -> (tactic * proof) -> reply
end

```

Figure 6.2: Interface of the Refinement Server

```
req_id * (tactic * proof).
```

We refer to pairs of tactics and proofs as *requests*. When we enqueue a request into the request queue, we assign it a unique identifier. These identifiers are represented as inhabitants of type `req_id`. There is only one request queue and it is mutually accessible by all ML threads. The functions for inserting and retrieving requests, which we describe later, provide mutually exclusive access to the request queue.

The reply table is a table containing members of type `reply` indexed by request ids. The type `reply` is implemented as the following datatype.

```
datatype reply =
  Failure of exn (* exn are SML exceptions *)
  | Success of proof list * (proof list -> proof)
```

The purpose of this datatype is to capture both possible outcomes of applying a tactic to a goal: A tactic application may raise an exception, or successfully terminate with a list of subgoals and a validation. Like the request queue, the reply table is accessible by all ML threads. Functions for inserting and retrieving replies, which we describe later, provide mutually exclusive access to the reply table.

The MP refiner initially contains one proc running the ML thread emulating a normal refiner. Additional procs are created using the function `spawn` (see Figure 6.2). The function `spawn` takes as an argument a *refinement server*, creates a proc, and runs the refinement server on the proc. A refinement server is a non-terminating ML function that polls the request queue. Refinement servers are inhabitants of type `server`. The function `server` whose interface is indicated in Figure 6.2 creates a refinement server. The integer argument of `server` is used to

create a time interval in which a refinement server waits before polling the request queue. This period of waiting is used to alleviate contention amongst procs for access to the request queue and the reply table. The second argument of `server` is a *request handler*. A request handler is a wrapper for a function representing the action to be taken when receiving a request. For our purposes, the action is tactic application using the tactic and goal within the request. The request handler catches all uncaught exceptions raised during the evaluation of the tactic application.

The functions `enqRequest` and `enqRequests` can be invoked during the evaluation of any ML expression to spawn the evaluation of one or more tactic applications. A request can be obtained from the request queue using the function `deqRequest`. Several requests can be dequeued simultaneously with the function `deqRequests`.

A reply is inserted into the reply table using the function `insertReply`. The three functions `getReply`, `waitForReply`, and `allReplies` are used to retrieve replies from the reply table. Applying `getReply` on a request id `reqId` results in the reply associated with the request id if it is in the table. The function `allReplies` obtains all the replies associated with a list of request ids; it terminates after it obtains all of the replies. While waiting for replies, `allReplies` polls the request queue for requests. Servicing requests while waiting for replies prevents deadlocks; deadlock may only occur when all the procs are blocked waiting for replies corresponding to requests in the request queue. The implementation of `allReplies` is given in Figure 6.3. The function `waitForReply` resembles `allReplies` except it

```

fun allReplies (r::r's) reqHandler = let
  val reqIds = (r::r's)
  fun action () =
    case deqRequest() of
      NONE => ()
    | SOME (reqId,req) =>
      insertReply (reqId,serviceRequest reqHandler req)
in
  case deqReply r of
    NONE => (action(); allReplies reqIds reqHandler)
  | SOME reply => reply :: allReplies r's reqHandler
end
| allReplies [] _ = []

```

Figure 6.3: Implementation of `allReplies`

waits for one reply.

```
fun waitForReply reqId = hd (allReplies [reqId])
```

The user can obtain the outcome of a tactic application embodied by a reply using the function `processReply`. If the tactic application raised an exception then `processReply` raises an exception. Otherwise, `processReply` returns the subgoals and validation produced by the tactic application. The function `discardReplies` removes replies from the reply table. Given a list of request ids, `discardReplies` removes all the replies associated with the request ids from the reply table.

The MP refiner supplies `RefinementServer` as a primitive ML structure so that users can create concurrent tactics. We use `RefinementServer` to implement concurrent tacticals. No tacticals are primitives of the MP refiner. Tacticals, like tactics, are user defined ML functions. We describe the implementation of concurrent tacticals in the following section.

## 6.2 Implementation of Concurrent Tacticals

A user creates concurrent tactics using the concurrent tacticals `PTHEN` and `PORELSEL`. `PTHEN` is constructed from the function `PThenOnEach`. `PThenOnEach` is derived from `THENOnEach` for creating the various forms of `THEN` in the standard tactic collection of Nuprl 4.1 created by Paul Jackson [39]. We provide the code of `PThenOnEach` in Figure 6.4. `PTHEN` is designed to implement AND-parallelism. The `else` portion of the `if` statement in `PThenOnEach` in Figure 6.4 implements AND-parallelism. The list of tactics, `tactics`, are implicitly passed as an argument to `PThenOnEach` through the function `extTacs`. The goals are obtained from the application of `tac` to `goal`. The application of the tactics in the list `tactics` to the goals in the list `subgoals` are packaged as requests to be performed by other refinement servers. `PThenOnEach` places all of the requests in the request queue and waits for all the results of the requests to be placed in the request queue. While waiting for the replies, `PThenOnEach` polls the request queue for request to service.

Notice that `PThenOnEach` spawns all of the tactic applications and retains none to perform itself. Because `PThenOnEach` uses `allReplies` to retrieve the results of the spawned applications, the refinement server may service some requests it placed in the request queue. When a refinement server services a request it spawned, in addition to performing the tactic application, time is wasted queuing the request, retrieving the request from the request queue, storing the reply in the reply table, and retrieving the reply from the reply table. To alleviate this problem, we can require `PThenOnEach` to only spawn some of the applications and carry out the rest itself. Using `PThenOnEach` we construct `PTHEN` as shown at the bottom of

```

(* some helper functions *)
fun applyTac (t,g) = t(g)
val reqHandler = RefinementServer.requestHandler applyTac
(*****)
fun PTHENOnEach (tac: tactic)
                (extTac: proof list -> tactic list)
                (goal:proof) =
let
    val (subgoals,validation) = tac goal
    val tactics = extTac subgoals
in
    if not (length tactics = length subgoals)
    then
        fail "PTHENOnEach: Wrong number of tactics"
    else let
        val reqIds =
            ListPair.map RefinementServer.enqRequests (tactics,subgoals)
        val replies = RefinementServer.allReplies reqIds reqHandler
        val results = RefinementServer.processReply replies
        val (newSubgoals,newValidations) = ListPair.unzip(results)
    in
        (List.concat newSubgoals,
         (validation o
          mapshape (map length newSubgoals) newValidations))
    end
end handle e => (print "PTHENOnEach: failing\n"; raise e)

fun PTHEN (t,t') = PTHENOnEach t (map (fn _ => t'))
infix PTHEN

```

Figure 6.4: Implementation of PTHEN using PTHENOnEach

```

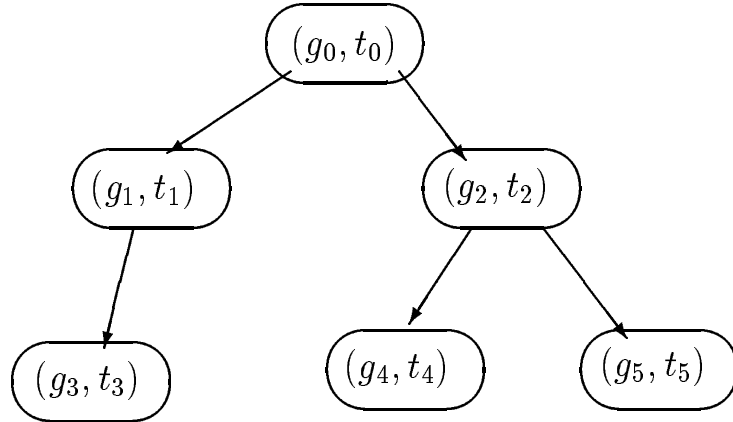
val applyTac (t,g) = t(g)
val reqHandler = RefinementServer.requestHandler applyTac
fun retrieveReplies (reqId :: reqIds) = let
  val reply = RefinementServer.waitForReply reqHandler reqId
in
  (processReply reply before discardReplies reqIds)
  handle _ => retrieveReplies reqIds
end
| retrieveReplies [] = fail "PORELSEL: all attempts failed"
fun PFirst (localTac::remoteTacs) goal = let
  val localTac = hd tactics
  val remoteTacs = tl tactics
  val requests = map (fn (t) => (t,goal)) remoteTacs
  val reqIds = RefinementServer.enqRequests requests
in
  (localTac(goal) before discardReplies reqIds)
  handle _ => retrieveReplies reqIds
end
fun PORELSEL (tac, tacs) = PFirst (tac::tacs)
infix PORELSEL

```

Figure 6.5: Implementation of PORELSEL using PFirst

Figure 6.4.

PORELSEL implements OR-parallelism by deterministically choosing a tactic from a list of tactics to use to refine a goal. We implement PORELSEL using the function PFirst. The code for PFirst is given in Figure 6.5. PFirst applies each tactic in the list `tactics` to the goal `goal`. The application of the first tactic in `tactics`, `localTac`, to `goal` is performed on the refinement server running PFirst. Before applying `localTac` to `goal`, PFirst spawns the application of the other tactics to `goal`. In Standard ML, `before` works like `;` except `before` returns the results of the first argument of `before` instead of the second argument. Therefore, PFirst, returns the result of the application of `localTac` to `goal` if it

Figure 6.6: Tactic Proof  $P$ 

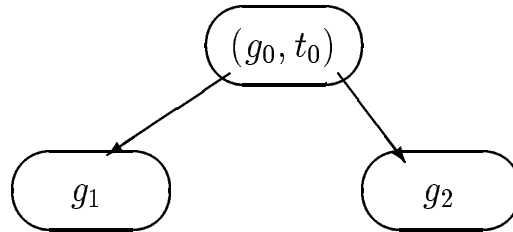
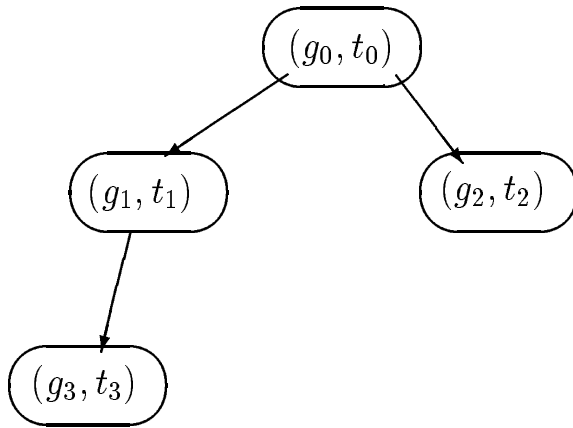
succeeds. If the application of `localTac` fails then an exception is raised causing the function `retrieveReplies` to be called. Using the function `retrieveReplies`, we return the result of the left-most tactic in `remoteTacs` that succeeds.

### 6.3 Non-determinism in Concurrent Refinement

Many tactic theorem provers support *replaying* proofs. Replaying a proof reconstructs the proof using the same tactics that created it. For example, to replay the proof  $P$  displayed in Figure 6.6, we do the following. First, we refine the goal in the root of  $P$ ,  $g_0$ , using the tactic  $t_0$ . Let  $P_1$  be the new tactic proof of  $g_0$  (see Figure 6.7).

Then we refine the leftmost unrefined goal in  $P_1$  using  $t_1$ . Let  $P_2$  be the new proof of  $g_0$  after the refinement. Then we refine the left most unrefined goal of  $P_2$  using the tactic  $t_2$ . This process continues until one of the following becomes true.

- All of the tactics in  $P$  have been used.

Figure 6.7: Tactic Proof  $P_1$ Figure 6.8: Finished replayed proof of  $P$  using less tactics

- A tactic fails to refine a goal.
- No unrefined goals exists.

The proof obtained from replaying is different from the original proof if all the tactics are not used. For example, suppose we change  $t_2$  after creating  $P$  so that refining  $g_2$  with  $t_2$  produces no subgoals. Then replaying  $P$  produces a complete proof of  $g_0$  containing fewer nodes than  $P$  (see Figure 6.8). Suppose we changed  $t_1$  instead of  $t_2$  so that  $t_1$  failed on  $g_1$ . Then replaying  $P$  produces an unfinished proof that resembles  $P$  except it does not have node  $\langle g_3, t_3 \rangle$ .

In the above examples, we obtained different replayed proofs of  $P$  after changing a tactic. However, we can obtain a different replayed proof without changing a tactic. If tactics are created using a programming language like ML, then it is possible that tactics contain references (see Section 2.1.7). For example, we defined  $t_2$  as follows.

```
fun t2 g =
  if !x = 0
  then
    fail
  else
    s(g)
```

Suppose that when  $t_2$  was used in creating  $P$  that  $!x = 1$ , and after creating  $P$ ,  $!x = 0$ . Replaying  $P$  will produce a different proof from  $P$ . The above shows that the ML state effects on the outcome of tactics. Actually, the entire refiner state effects the outcome of a tactic. For example, suppose that one of the steps of primitive refinement performed by  $t_1$  uses the primitive refinement rule  $r$ . If  $r$  is removed from the primitive refinement rule table, then  $t_1$  will fail to perform the step of primitive refinement using  $r$ .

In general, two different evaluations of a tactic will be the same if both evaluation are performed in the context of the same refiner state. We state this property as the following lemma.

**Lemma 6.3.1** The outcome of refining a goal  $g$  using a tactic  $t$  with respect to a refiner state  $R$  is unique.

**Proof:** The proof of this lemma boils down to showing that the value produced from evaluating an ML expression within the same environment is unique.

□

From this Lemma 6.3.1, we deduce the following theorem regarding replaying.

**Theorem 6.3.2** Let  $P$  be a tactic proof developed in the context of refiner state  $R$ . Suppose that the tactics in  $P$  have not be changed since  $P$  was constructed. Let  $P'$  be the replayed proof of  $P$  obtained in the context of the refiner state  $R'$ . Then  $P \neq P'$  implies  $R \neq R'$ .

We note that Theorem 6.3.2 holds only for proofs containing sequential tactics. In other words, if a proof  $P$  contains one or more concurrent tactics and is constructed in the context of refiner state  $R$ , then replaying  $P$  in the context of refiner state  $R$  can result in a different proof. This phenomena occurs when a concurrent tactic spawns at least two tactic applications. One of the tactic application updates a reference that other tactic application uses to determine its outcome. For example, let  $c$  be the concurrent tactic ( $t \text{ PORELSEL}[t']$ ) where  $t$  and  $t'$  are given below.

```

fun t pf =
  (x := 1;
   if !x = 1
     then fail
     else s(pf))
fun t' pf = (x := 0; s'(pf))

```

Based on the semantics of PORELSEL, evaluating  $c(p)$  results in evaluating  $t(p)$  and  $t'(p)$  concurrently. As a result, the outcome of  $c(p)$  depends on the sequential ordering of the evaluations of  $t(p)$  and  $t'(p)$ . For example, suppose  $x := 0$  in  $t'$  is evaluated before  $x := 1$  in  $t$ , then  $c(p)$  evaluates to the result of  $s'(p)$ . However, suppose  $x := 0$  in  $t'$  is evaluated after  $x := 1$  in  $t$ . Then  $c(p)$  evaluates to the result of  $s(p)$ . Thus  $c$  has a *non-deterministic* behavior.

```

datatype 'a rref = rref of tactic_id * 'a ref

fun !! (rref (tacId, refCell)) = !refCell
fun ::= (rref (tacId, refCell), value) =
  if tacId = idOfThread ()
  then refCell := value
  else
    fail "No permission to assign to ref cell"
infix ::=

```

Figure 6.9: Definition of Restricted References

To eliminate non-determinism, we could prevent the use of references within concurrent tactics. Although this solution works, it is too restrictive. Instead, we prevent certain threads from updating a reference. Specifically, a reference may be modified only by the thread that created it. The evaluation of a spawned tactic application is a thread of ML computation. Recall from Chapter 6 that a request id is associated with each spawned tactic application. We can use the request id to represent the *tactic thread id*. The declarations in Figure 6.9 define references that allow any thread to access the reference, but only the thread that created the reference can modify it. In Figure 6.9, `tactic_id` is the type of tactic thread ids. The function `!!` dereferences a restricted reference, and `::=` updates a restricted reference. If we used restricted references in the definitions of `t` and `t'` above, then the tactic `(t PORELSEL [t'])` would always fail. The tactics `f` and `f'` defined below demonstrate using restricted references.

```

fun f pf =
  if !!y = 1
  then fail
  else s(pf)

```

```
fun f' pf = let
  val x = rref 1
in
  x ::= 0;
  (s1 PORELSEL [s2]) pf
end
```

By evaluating  $(f \text{ PORELSEL } [f'])(p)$ , two threads are created, one executing  $f(p)$  and the other executing  $f'(p')$ . Because  $y$  was defined before the execution of  $f(p)$ , an exception is raised if it attempts to update  $y$ . When executing  $f'(p)$ , two additional threads are spawned, one executing  $s1(p)$  and the other executing  $s2(p)$ . When executing  $s1(p)$  and  $s2(p)$ ,  $x$  may be accessed, but it cannot be modified.

# Chapter 7

## Performance Analysis of Concurrent Refinement

In this chapter, we report on tests with the MP refiner to determine whether concurrent refinement improves the performance of tactics. We use the cost model in Section 7.1 to identify a class of tactics that may benefit from concurrent refinement. In Section 7.2, we describe the methods used to conduct our tests. In Section 7.3, we present the results of the test and discuss the outcomes in Section 7.4.

### 7.1 Cost Model of Tactic Applications

We model the execution cost of a tactic as the elapsed time to perform all the primitive inferences invoked by the tactic. The cost excludes time spent performing computations other than refinements. In practice, the costs of two primitive refinements can significantly differ. One primitive refinement may invoke an expen-

sive decision procedure while the other only performs matching and substitution . For our cost model, however, we assume that all primitive inferences have fixed unit cost. Thus, the cost of the application of a sequential tactic to a goal is the number of primitive refinements the tactic successfully invokes. Therefore, a tactic that fails on a goal may have a positive cost: It may invoke some primitive refinements successfully, and then at some point it invokes a primitive refinement that fails on the goal. Thus, if  $t_i$  fails on  $g$  for  $0 \leq i < j \leq n$ , and the result of applying

$$t_0 \text{ ORELSEL } \{t_1, \dots, t_n\} \tag{7.1}$$

on  $g$  is  $t_j(g)$ , then the cost of applying (7.1) to  $g$  is the sum of the costs of applying each  $t_i$  to  $g$  plus the cost of applying  $t_j$  to  $g$ . The cost of applying ( $t$  THEN  $t'$ ) to a goal  $g$  is the cost of applying  $t$  to  $g$  plus the sum of the cost of applying  $t'$  to each subgoal produced from applying  $t$  to  $g$ .

The cost of concurrent tactics has to compensate for the overhead of placing a tactic application in the request queue, retrieving a tactic application from the request queue, placing the result of a tactic application in the reply table, and retrieving a result from the reply table. The request queue and the reply table are shared data structures requiring mutually exclusive access. To determine the cost of the overhead caused by spawning tactic applications, we assume that there is no contention in accessing the request queue and reply table. Therefore, we reflect the overhead of enqueueing and dequeuing as a constant. In addition to overhead, cost of a concurrent tactic must reflect the amount of time requests wait in the request queue before being serviced. For example, suppose applying  $t$  to  $g$  produces  $n$  subgoals and there are  $k$  refinement servers where  $k < n$ . Then

as a result of applying  $(t \text{ PTHEN } t')$  to  $g$ , there will be more requests generated than there are refinement servers. Thus, some of the requests will be sitting in the request queue, while others are being executed. If  $M$  is the maximum cost of all  $n$  tactic applications, then we can express the cost of the  $n$  tactic applications as

$$M \left\lceil \frac{n}{k} \right\rceil.$$

We formally define the cost of a tactic application as follows.

**Definition 7.1.1** Let  $t$  be a tactic and  $g$  a goal. Let  $k > 1$  be the number of processors. Let  $\alpha > 0$  be a constant representing the overhead for spawning a single tactic application and retrieving its results. If  $t$  is a simple tactic then  $\text{cost}_g(t)$  is the number of primitive refinements successfully invoked by applying  $t$  to  $g$ . If  $t = (t' \text{ THEN } t'')$ , then

$$\text{cost}_g(t) = \text{cost}_g(t') + \text{cost}_{g_1}(t'') + \cdots + \text{cost}_{g_n}(t'')$$

where  $g_1, \dots, g_n$  are the unrefined goals produced when applying  $t$  to  $g$ . If  $t$  fails on  $g$ , then  $\text{cost}_g(t) = \text{cost}_g(t')$ .

If  $t = (t_0 \text{ ORELSEL } [t_1, \dots, t_n])$ , then

$$\text{cost}_g(t) = \text{cost}_g(t_0) + \text{cost}_g(t_1) + \cdots + \text{cost}_g(t_j)$$

where either  $j = n$  and  $t_i$  fails on  $g$  for  $i = 1, \dots, n - 1$ , or  $j < n$ ,  $t_j$  succeeds on  $g$ , and  $t_i$  fails on  $g$  for  $i = 1, \dots, j - 1$ .

If  $t = (t_0 \text{ PORELSEL } [t_1, \dots, t_n])$ , then

$$\text{cost}_g(t) = \max(\text{cost}_g(t_0), \dots, \text{cost}_g(t_n)) \left\lceil \frac{n}{k} \right\rceil + \alpha n$$

If  $t = (t' \text{ PTHEN } t'')$ , then

$$\text{cost}_g(t) = \text{cost}_g(t') + \max(\text{cost}_{g_1}(t''), \dots, \text{cost}_{g_n}(t'')) \left\lceil \frac{n}{k} \right\rceil + \alpha n$$

where  $g_1, \dots, g_n$  are the unrefined goals produced from applying  $t'$  to  $g$ . If  $t'$  fails on  $g$  then  $\text{cost}_g(t) = \text{cost}_g(t')$ .

Using our cost model, we may reason about the speedup of concurrent tactics over sequential tactics. For example, from our cost model we can determine that the sequential tactic  $(t \text{ THEN } t')$  out performs the concurrent tactic  $(t \text{ PTHEN } t')$  if  $t$  produces a few subgoals, and the cost of  $t'$  on each of the subgoals is low with respect to  $\alpha$ . We state this as the following lemma.

**Lemma 7.1.2** Let  $s = (t \text{ THEN } t')$ ,  $s' = (t \text{ PTHEN } t')$ , and  $g$  be a goal. Let  $k$  be the number of processors. Suppose  $t$  produces the unrefined goals  $g_1, \dots, g_n$  when applied to  $g$  and  $n \leq k$ . If the average cost of the tactic applications  $t'(g_1), \dots, t'(g_n)$  is less than or equal to  $\alpha$ , then  $\text{cost}_s(g) \leq \text{cost}_{s'}(g)$ .

**Proof:** The costs of the tactics  $s$  and  $s'$  when applied to  $g$  are

$$\begin{aligned} \text{cost}_s(g) &= \text{cost}_t(g) + \text{cost}_{t'}(g_1) + \dots + \text{cost}_{t'}(g_n) \\ \text{cost}_{s'}(g) &= \text{cost}_t(g) + \max(\text{cost}_{t'}(g_1), \dots, \text{cost}_{t'}(g_n)) \left\lceil \frac{n}{k} \right\rceil + \alpha n. \end{aligned}$$

Assume without loss of generality that

$$\max(\text{cost}_{t'}(g_1), \dots, \text{cost}_{t'}(g_n)) = \text{cost}_{t'}(g_1).$$

Then  $\text{cost}_s(g) \leq \text{cost}_{s'}(g)$  if and only if

$$\text{cost}_t(g) + \text{cost}_{t'}(g_1) + \dots + \text{cost}_{t'}(g_n) \leq \text{cost}_t(g) + \text{cost}_{t'}(g_1) + \alpha n$$

$$\frac{\text{cost}_{t'}(g_2) + \cdots + \text{cost}_{t'}(g_n)}{n} \leq \alpha$$

Hence, if  $\alpha$  is greater than or equal to the average cost of the tactic applications  $t'(g_1), \dots, t'(g_n)$ , then  $\text{cost}_s(g) \leq \text{cost}_{s'}(g)$ .

□

We use our model of cost of tactic applications to establish the following lemma regarding the potential speedup of concurrent tactics over sequential tactics.

**Lemma 7.1.3** The cost of the sequential tactic ( $t_0$  ORELSEL  $[t_1, \dots, t_n]$ ) is less than the cost of the concurrent tactic ( $t_0$  PORELSEL  $[t_1, \dots, t_n]$ ) if  $t_0$  succeeds.

## 7.2 Methods

We measure the comparisons of the running times of sequential tactics and concurrent tactics as *speedup*. Speedup is the ratio of the running time of the sequential program to the parallel program. The parallel program is a parallel implementation of the algorithm implemented by the sequential program. A speedup of  $k$  for  $k > 1$ , for example, means the parallel program is  $k$  times faster than the sequential program. When the sequential program implements the most optimal algorithm for a task, then the greatest theoretical speedup, *linear speedup*, is equal to the number of processors executing the parallel program. Therefore, a parallel program significantly out performs a sequential program if it has near linear speedup.

The additional overhead of executing a parallel program, such as main–memory bus contention and interprocessor communication, effects the performance of par-

allel programs. Therefore, we use *efficiency* [49] to measure the ability of a parallel program to make productive use of multiple processors. Efficiency is the ratio of the speedup to the number of processors. An efficiency close to one means that overhead has little effect on the execution of the parallel program. An efficiency close to zero means that overhead has significant effect on the execution of the parallel program.

To calculate the speedups, we obtain the running times of sequential tactics on the SML implementation of the refiner. Our sample of tactics were created using THEN and ORESLEL. The tactics used as arguments to THEN and ORESLEL are listed in Table 7.1 and Table 7.2. The running time of each tactic in Table 7.1 and Table 7.2 is the average running time of 25 runs. The running times were obtained using the SML refiner.

We use the following naming convention for our tactics. For a tactic `tac $i$ _ $j$` , where  $i$  and  $j$  are integers,  $i$  is the number of primitive refinements the tactic performs and  $j$  is the number of unrefined subgoals generated. For example, `tac65_10` performs 65 primitive refinements and generates 10 subgoals. The tactics with the names `tac1_ $j$`  were created using the primitive refinement rule `rule $j$`  that generates  $j$  subgoals. Given any goal  $g$ , `rule $j$`  generates  $j$  subgoals, each equal to  $g$ .

The tactics created with THEN used each of the tactics in Table 7.1 as the left hand side argument of THEN. We chose these tactics based on Lemma 7.1.2. In preliminary tests, we also used `tac1_1` as a tactic on the left hand side of THEN, but these tactics out performed their concurrent counterparts. (See Appendix E

Table 7.1: Tactics on the Left Hand Side of THEN

<b>Tactic</b>	<b>Running Time (Seconds)</b>
tac1_10	0.006
tac1_50	0.022
tac1_100	0.047

Table 7.2: The Right Hand Side Tactics

<b>Tactic</b>	<b>Running Time (Seconds)</b>
tac1_1	0.002
tac65_10	0.01
tac400_100	0.649
tac2000_500	3.379
tac10000_100	16.567

for the results of the preliminary tests.) The tactics used on the right hand side of THEN appear in Table 7.2.

Because of Lemma 7.1.3, we created tactics with ORELSEL for which only the last alternative succeeded.<sup>1</sup> We only used one tactic as the left hand side tactic, namely tac65\_10. The tactics used as alternatives are listed in Table 7.2. We used

---

<sup>1</sup>In preliminary tests, tactics created with ORELSEL in which the first tactic succeed out performed there concurrent versions. See Appendix for details.

```

fun ORELSELF (tac, []) = (FailT tac) ORELSEL []
  | ORELSELF (tac, tacs) = let
    val (lastTac::tail) = rev tacs
    val failingTacs = map FailT tacs
  in
    (FailT tac) ORELSEL (rev (lastTac :: failingTacs))
  end
infix ORELSELF

```

Figure 7.1: Definition of ORELSELF

each tactic in Table 7.2 to create lists containing 1, 5, 20, and 50 alternatives. We used each list as the right hand side argument of ORELSEL. However, to create the set of tactics for which only the last alternative succeeds, we applied the tactical FailT to each tactic except the last alternative:

```

fun FailT tactic pf = (tactic pf; fail "Tactic Failed")

```

For example,

```

(FailT tac65_10) ORELSEL [FailT tac1_1, FailT tac1_1,
                          FailT tac1_1, FailT tac1_1, tac1_1]

```

is a tactic from our sample in which only the last alternative succeeds. To simply creating tactics where only the last alternative succeeds, we used the tactical ORELSELF (pronounced or-else-l-f). ORELSELF is the same as ORELSEL except ORELSELF automatically applies FailT to the appropriate arguments of ORELSEL to ensure only the last alternative succeeds (see Figure 7.1).

For each sequential tactic, we created a concurrent tactic by replacing THEN with PTHEN, and ORELSELF with PORELSELF in the sample of sequential tactics. We obtained the running times of each concurrent tactic executing on the MP refiner with two, three, and four refinement servers. The MP refiner ran on a Sun

Sparc 670 shared-memory multi-processor with four processors. We obtained the times using the timer supplied by SML/NJ. The timer uses `gettimeofday` to obtain the current time.

According to [43], a parallel program is scalable if the efficiency of the parallel program remains constant as the problem size and the number of processors increase. For example, suppose we have a parallel program that runs on a hypercube. The number of addends of the parallel program constitutes the problem size. To determine scalability, we examine the efficiencies of the program for 4, 8, 16, 32, and 64 addends on a hypercube with 2, 4, 8, 16, and 32 processors, respectively. In our setting, we have two problems, AND-search and OR-search. AND-search is tactic refinement using a tactic created with `THEN` or `PTHEN`. OR-search is tactic refinement using a tactic created with `ORELSEL` or `PORELSEL`. Increasing the problem size of AND-search and OR-search involves increasing the number of branches to traverse. For example, replacing `tac1_10` with `tac1_100` in the tactic

```
tac1_10 PTHEN tac65_10
```

increases the problem size of the AND-search performed by `tac65_10`. Increasing the number of alternatives on the right of `PORELSEL` to 10 in

```
tac65_10 PORELSEL [tac2000_500, tac2000_500, tac2000_500]
```

increases the problem size of OR-search performed by `tac2000_500`. In our tests, the problem sizes for AND-search are 10, 50, and 100. When presenting the speedups and efficiencies of AND-parallelism using the tactic  $t$ , we display the speedups and efficiencies of

tac1\_10 PTHEN  $t$ ,  
 tac1\_50 PTHEN  $t$ , and  
 tac1\_100 PTHEN  $t$

using two, three, and four refinement servers, respectively. Likewise, when presenting the speedups and efficiencies of OR-parallelism using the tactic  $t$ , we display the speedups and efficiencies of

tac65\_10 PORELSELF  $\bar{t}_5$ ,  
 tac65\_10 PORELSELF  $\bar{t}_{20}$ , and  
 tac65\_10 PORELSELF  $\bar{t}_{50}$

using two, three, and four refinement servers, respectively. For a tactic  $t$ ,  $\bar{t}_i$  is the list containing  $i$  copies of  $t$ .

### 7.3 Results

Our results show that concurrent refinement modestly improves the running times of tactics. Figure 7.2 depicts speedups of tactics using AND-parallelism. Figure 7.3 depicts speedups of tactics using OR-parallelism. None of the tactics achieve linear speedups; the closest we get is 2.825, the speedup of

tac1\_100 PTHEN tac65\_10 (7.2)

using four refinement servers. The speed of 2.825 indicates that (7.2) on four processors is almost three times as fast as

tac1\_100 THEN tac65\_10. (7.3)

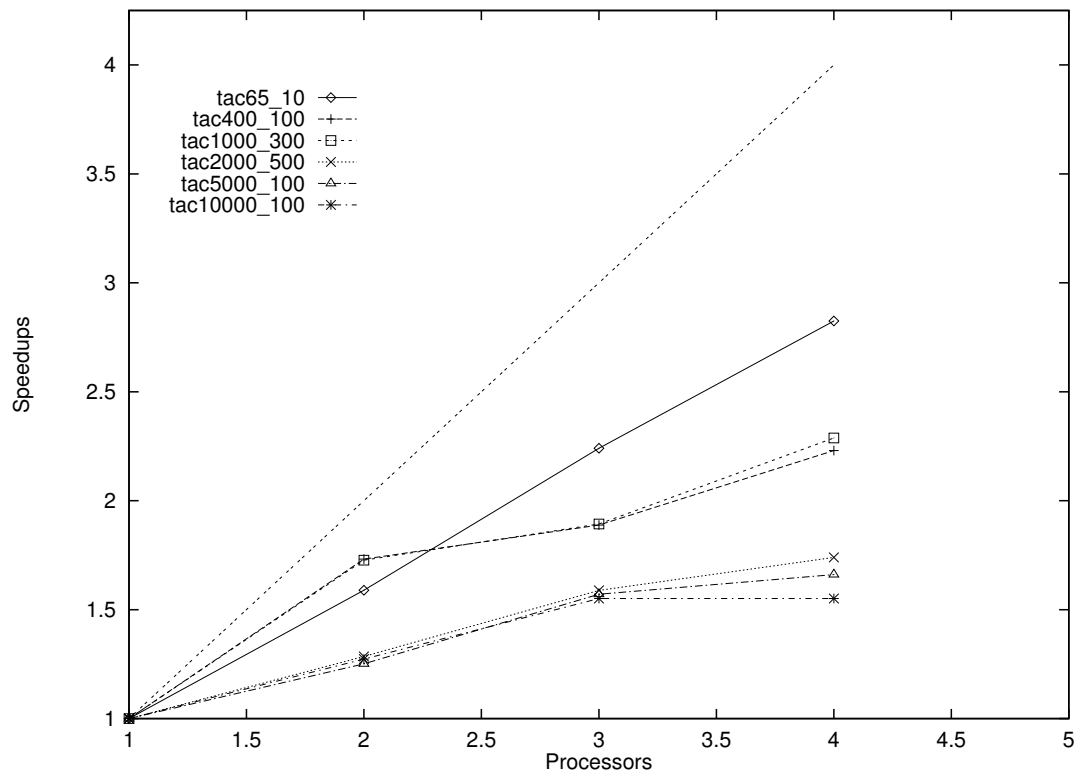


Figure 7.2: AND-parallelism Speedups

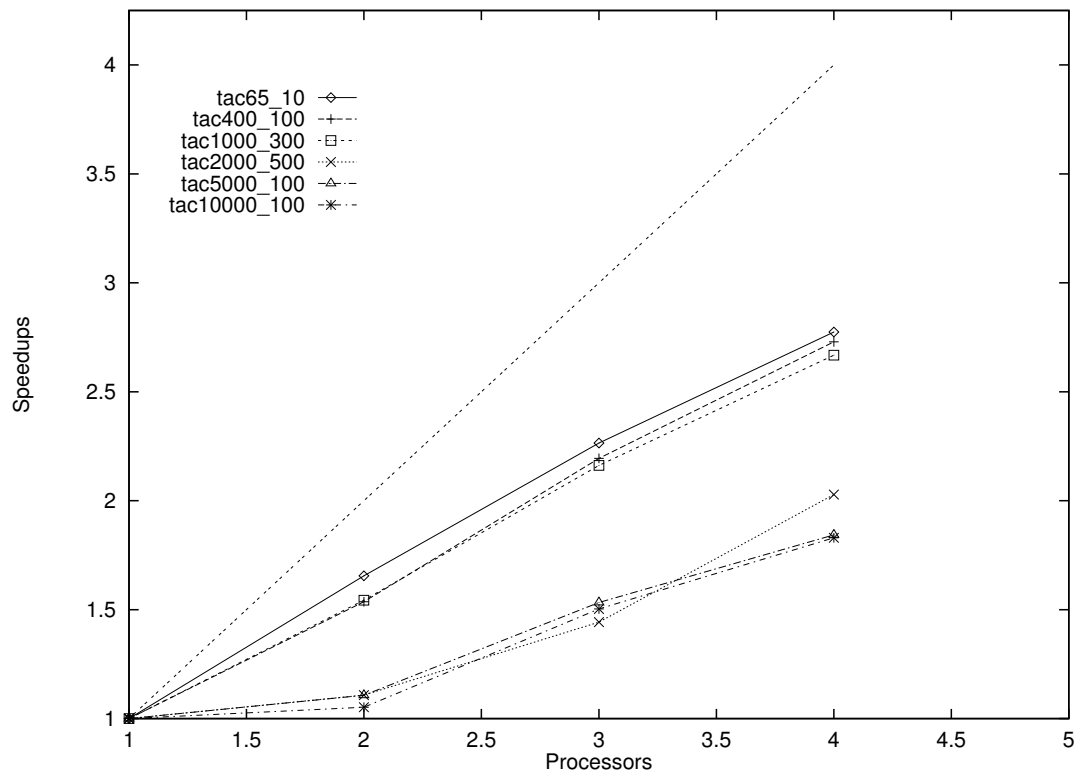


Figure 7.3: OR-parallelism Speedups

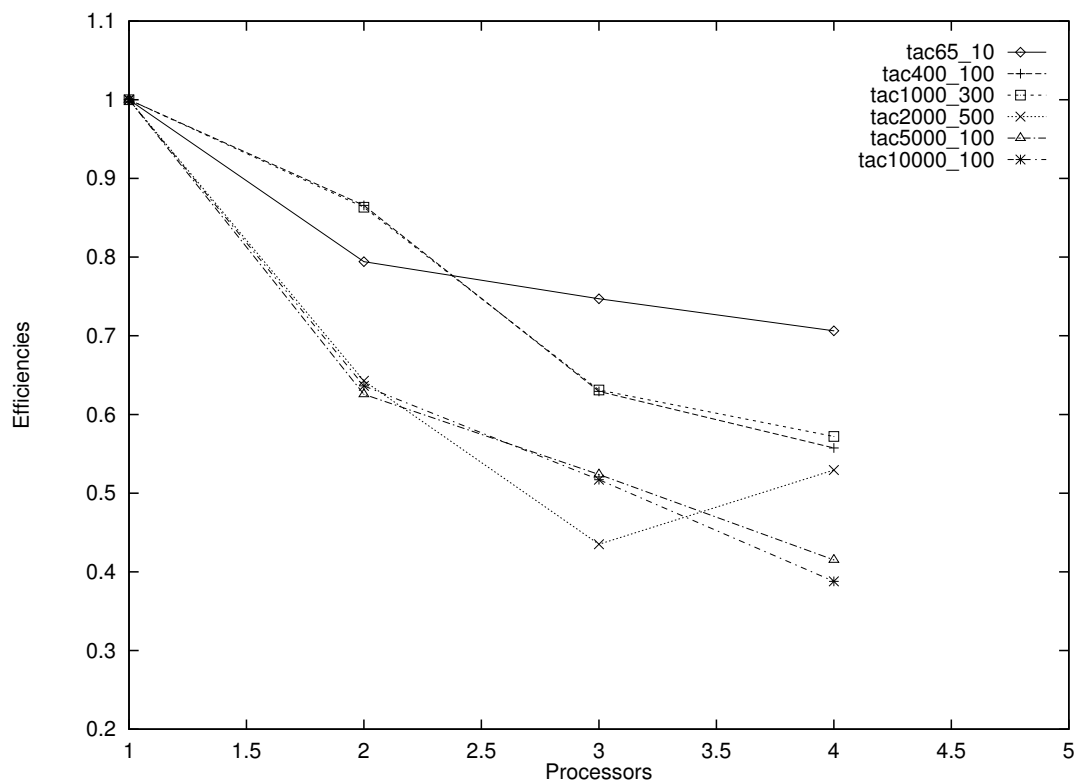


Figure 7.4: AND-parallelism Efficiencies

The continually decreasing efficiencies in Figure 7.4 and Figure 7.5 reveal that the MP refiner is not scalable. However, the MP refiner scales better with respect to OR-parallelism than AND-parallelism.

## 7.4 Discussion

We obtained significant speedups primarily using AND-parallelism and OR-parallelism with `tac65_10`. This shows that the MP refiner works well using a fine-grain of parallelism. The running time of `tac65_10` is roughly the average running time of a refinement rule in Nuprl 4.1. Performance tests conducted by Rich Eaton show that in Nuprl 4.1 the average running time of a refinement rule is

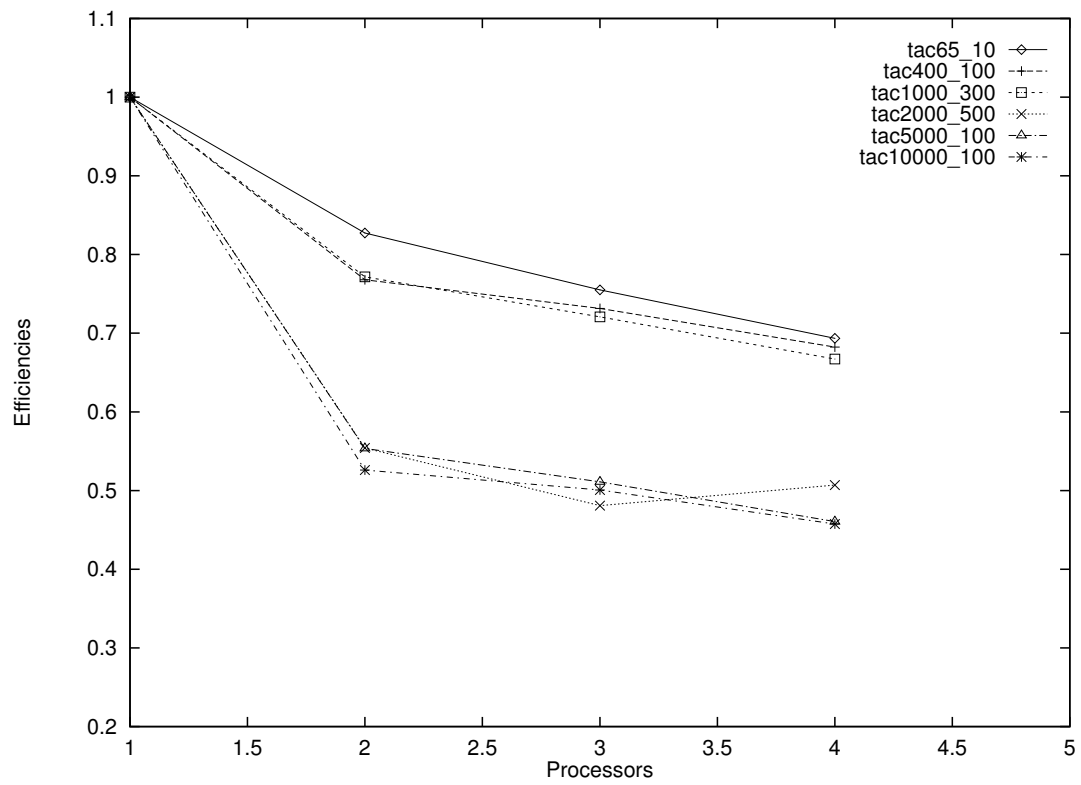


Figure 7.5: OR-parallelism Efficiencies

0.012 seconds. The running time of `tac65_10` is 0.1 seconds. Thus significant speedups are possible by employing parallelism at the refinement rule level in Nuprl. Originally we expected that parallelism would be more beneficial for tactics such as `tac10000_100`. However our results show that the MP refiner performed poorly on such large tactics.

We believe that memory management of the MP runtime system causes the MP refiner to perform poorly for large tactics. Morrisett and Tolmach [50] conjectured that contention between processors for the main-memory bus hinders performance in the MP runtime system. Morrisett and Tolmach used five benchmarks to test the performance of the MP runtime system. Of the five benchmarks, only one experienced near linear speedup<sup>2</sup>. This benchmark also generated the least bus traffic. The other benchmarks generated a high volume of bus traffic caused by successive cache misses. Morrisett and Tolmach believe the successive cache misses resulted from poor locality of reference exhibited by the SML/NJ strategy for allocating memory. We noticed in our tests that the tactics requiring the least memory had the greatest speedups: compare the speedups of (7.4), 2.825, and (7.5), 1.551.

$$\text{tac1\_100 PTHEN tac65\_10} \tag{7.4}$$

$$\text{tac1\_100 PTHEN tac10000\_100} \tag{7.5}$$

The application of (7.4) to a goal requires less memory than the application of (7.5) to a goal. During a tactic application, the tactic creates a proof. A refinement

---

<sup>2</sup>Actually, the benchmark had super-linear speedup.

server needs memory to store the proof. The size of the proof depends on the number of successful primitive refinements invoked by the tactic.

The MP refiner may behave poorly for large tactics because of contention between refinement servers for the request queue and the reply table. However, we do not believe contention between refinement servers significantly hinders performance of concurrent tactics. Recall from Chapter 6 that the request queue and the reply table are mutually exclusive. Therefore, refinement servers may contend for access to the request queue and the reply table. The probability of contention is based on the number of requests and the number of refinement servers. Thus contention should equally effect (7.4) and (7.5), but their speedups differ drastically.

Large tactics may suffer from the synchronization of procs to perform garbage collection. Recall from Section 2.3.1 that only one proc performs garbage collection in the MP runtime system while others busy wait. As a result, the MP runtime system is unable to parallelize a portion of the sequential computation. For large tactics, however, the percentage of overhead due to parallel computation significantly surpasses the percentage of computation the sequential tactics devote to garbage collection. For example, garbage collection accounts for 11% of the computation of

$$\text{tac1\_100 THEN tac10000\_100.} \quad (7.6)$$

However, overhead due to parallel computation accounts for 61% of the computation of (7.5).

One way of improving the performance of the MP refiner is to implement it on a loosely-coupled multi-processor. As a result, the MP refiner would be a col-

lection of processes running on separate processors. The processes are sequential refiners that communicate by passing messages. We initially designed the MP refiner for a loosely-coupled multi-processor, but decided not to implement it because of the difficulty communicating the results of spawned tactic applications. We intended to spawn tactic application using *distributed evaluation*. Distributed evaluation describes the process of compiling source code into intermediate code on one machine and transmitting the intermediate code to another machine. The other machine translates the intermediate code into machine code and executes the machine code. Java uses distributed evaluation to support active Web documents [26]. Spawning a tactic application constitutes compiling the tactic into an SML/NJ lambda expression and having another refiner translate the lambda expression into machine code and evaluate it [41]. The result of the tactic application had to be transmitted to the refiner that spawned the application and placed on its heap. In SML, programmers may only place an object on the heap by evaluating a declaration. Recall from Section 5.3 that the result of a tactic application is a list of goals and a validation. Therefore, returning the result of the tactic application requires translating the goals and validation into an object that can be evaluated by another refiner as a declaration of a list of subgoals and a validation. This process may require a great deal of overhead in addition to the overhead of inter-processor communication. A possible solution requires only returning the subgoals. The contents of the subgoals are much easier to marshall as compared to a validation: A validation is basically a proof and it can be extremely large. This solution, however, changes the semantics of tactic refinement.

We intend to implement the MP refiner to run on a loosely-coupled multiprocessor as future work.

# Chapter 8

## Conclusion

In this dissertation, we described the implementation of a prototype of the first parallel interactive theorem prover. Our prover, the MP refiner, uses AND-parallelism and OR-parallelism on a shared-memory multi-processor. The MP refiner is a multi-processor implementation in SML of the Nuprl refiner. The MP refiner makes significant improvement on the throughput of tactics when using a fine-grain of parallelism and only modest improvements using a course-grain of parallelism. Granularity is measured in the number of successful primitive refinements invoked by a tactic. We believe the modest improvements with course-grain parallelism occur because of contention between the processors for the main-memory bus due to the memory management of SML/NJ. Concurrent tactics employing course-grain parallelism have higher memory requirements than concurrent tactics employing fine-grain parallelism.

As a by-product of re-implementing the Nuprl refiner in SML, we developed a design of the refiner. A formal design of a software system consists of a for-

mal specification of the system, a listing of the components of the system, and a description of the relationships of the components [28]. Instead of a formal specification, we developed a rigorous mathematical description of the refiner, a model of tactic refinement independent of any programming language, and a rigorous description of the Nuprl second order substitution algorithm. The model of tactic refinement covers sequential and concurrent refinement. The mathematical description is based on the reflected Nuprl type theory [3,5,4] and abstract refinement [32]. The description of the components of the refiner and their relationships is the first written account of the design of the Nuprl 4 refiner.

## **Future Work**

In the future, we intend to implement the MP refiner as a loosely-coupled multi-processor (see Section 7.4). Implementing the MP refiner on a loosely-coupled multi-processor would make the MP refiner a collection of refiners executing on separate processors. Therefore, we would eliminate the contention with the main-memory bus and synchronizing all the processors for garbage collection. However, we would introduce overhead of passing messages between separate ML processes. In Section 7.4, we described the problems that arise from passing messages between ML processes.

We also intend to investigate implementing the MP refiner using an ML compiler other than SML/NJ. A different compiler may produce programs with less memory requirements, thereby, reducing bus contention. However, re-implementing the MP refiner using another ML compiler may be a major undertaking. Using another compiler may require rewriting a large part of the MP refiner. In addi-

tion, we may need to enhance the compiler to exploit parallelism. Instead, we could incorporate concurrent refinement in another implementation of Nuprl using a different compiler other than SML/NJ. For example, we could incorporate concurrent refinement in Nuprl-light [35]. Nuprl-light is an implementation of Nuprl in Caml-Light that uses an object-oriented framework for specifying theories.

In the future, we intend to experiment with pipelining in the MP refiner. Pipelining is concurrent refinement using multiple compositions of PTHEN. To understand pipelining, consider applying the tactic  $(t \text{ THEN } (t' \text{ THEN } t''))$  to the goal  $g$ . First, the tactic  $t$  is applied to  $g$  to produce the subgoals  $g_1, \dots, g_n$ . Afterward,  $t'$  is applied to each  $g_i$  producing the subgoals  $g_{i,1}, \dots, g_{i,m_i}$ . Pipelining involves applying  $t''$  to  $g_{1,1}, \dots, g_{1,m_1}$ , for example, and  $t'$  to  $g_2$  simultaneously. Multiple compositions of PTHEN, for example  $(t \text{ PTHEN } (t' \text{ PTHEN } t''))$ , implements pipelining.

Additional future work includes creating a parallel interactive theorem prover using a master-slave design. The master prover is a typical interactive theorem prover that uses one or more automatic theorem provers as slaves. The master prover spawns goals for the slaves to completely prove automatically. The master prover only spawns goals it believes the slave can prove automatically. While the slaves are proving the goals, the master continues with its own work. This approach would probably be more favorable to interactive theorem provers such as IMPS or PVS that employ decision procedures as part of the inference strategy. Nuprl can also use this type of structure for proving well-formedness goals. Well-formedness goals are generated by the Nuprl type theory refinement rules to validate that a

term is a type. These goals usually can be proven automatically by the `Auto` tactic [39]. A slave prover simply refines all goals it receives using `Auto`.

# Appendix A

## Parameters

We start with an infinite countable number of identifiers and variable symbols denoted as the types  $\mathcal{I}$  and  $\mathcal{U}$ , respectively. A triple  $\langle S, O, \approx \rangle$  is a *primitive object structure* if it meets the following criteria.

1.  $S$  is a type.
2.  $O$  is a finite type and  $\forall f \in O \ f \subseteq S^n$  for some  $n \geq 1$ .
3.  $\approx \subseteq S \times S$  is an equivalence relation.

$S$  is the *domain* of  $\langle S, O, \approx \rangle$ . The elements of  $O$  represent predicates, operations, and relations over  $S$ . An example of a primitive object structure is

$$\langle \mathcal{N}, \{+, \times, -, <, \text{mod}, \text{div}\}, = \rangle.$$

For primitive object structures  $(\Sigma_1, \dots, \Sigma_n)1k$  and  $\{(n_1, \dots, n_n)1k\} \subseteq \mathcal{I}$ , we define an *indexed family* as the set  $\{\langle n_1, \Sigma_1 \rangle, \dots, \langle n_k, \Sigma_k \rangle\}$ . Given an indexed family  $\Gamma$ , a *parameter* is a pair  $\langle p, t \rangle$ , written  $p : t$ , if  $\langle t, \langle S, O, \approx \rangle \rangle \in \Gamma$  and either  $p \in \mathcal{U}$  or  $p \in S$ . We say  $p$  is the value of the parameter and  $t$  is the tag of the parameter.

We would like to fix  $\Gamma$  to a specific index family. The objects which we would want to be primitive in the Nuprl term language are variables, strings, natural numbers, booleans, and level expressions. Level expressions are used as indexes for universes in the hierarchy of type universes [58]. We denote them as the set  $L$ . The set  $L_O$  contains the predicates, operations, and relations for level expressions. We denote equality over  $L$  as  $L_\approx$ . We describe  $L$ ,  $L_O$ , and  $L_\approx$  formally in Appendix B. The primitive object structure for level expressions is  $\Sigma_L = \langle L, L_O, L_\approx \rangle$ . The primitive object structure for the variables is  $\Sigma_U = \langle \mathcal{U}, \mathcal{U}_\approx \rangle$  where  $\mathcal{U}_\approx$  is the obvious equality over  $\mathcal{U}$ . The primitive object structure for strings is  $\Sigma_S = \langle S, \{\cdot\}, S_\approx \rangle$  where  $S$  is the set of sequences of printable characters,  $\cdot$  is concatenation, and  $S_\approx$  is the obvious equality over  $S$ . We let  $\Sigma_B = \langle \{\text{tt}, \text{ff}\}, \{\wedge, \vee, \neg\}, B_\approx \rangle$  be the primitive object structure for booleans. The primitive object structure for naturals is  $\Sigma_N = \langle \mathcal{N}, \{+, \times, -, <, \text{mod}, \text{div}\}, = \rangle$ . We set  $\Gamma$  to be the indexed family

$$\{\langle l, \Sigma_L \rangle, \langle v, \Sigma_U \rangle, \langle s, \Sigma_S \rangle, \langle b, \Sigma_B \rangle, \langle n, \Sigma_N \rangle\}.$$

# Appendix B

## Level Expression Predicates, Operations, and Relations

This presentation on level expression was taken from unpublished documentation written by Doug Howe with changes by Paul Jackson with respect to level constants.

Level expressions are used as indexes to type universes [72]. We define level expressions inductively as the type  $L$  below.

1. (level variables) If  $u \in \mathcal{U}$  then  $u \in L$ .
2. (level constants) If  $c \in \{n \in \mathcal{N} \mid n > 0\}$ , then  $c \in L$ .
3. (increments) If  $l \in L$  and  $k \in \mathcal{N}$  then  $\langle l, k \rangle \in L$  written as  $lk$ . If  $k = 1$ ,  $lk$  is written as  $l'$ .
4. (maximums) If  $l_1, \dots, l_n \in L$  then the sequence  $(l_1, \dots, l_n) \in L$  and is written as  $l_1 | \dots | l_n$ .

A *level assignment* is a function  $v$  from level variables to level constants. We extend a level assignment  $v$  to level expressions by  $v(c) = c$  for  $c$  a level constant,  $v(en) = v(e) + n$ , and  $v(e_1 | \cdots | e_k) = \max(v(e_1), \dots, v(e_k))$ . For level expressions  $e_1$  and  $e_2$ , we define the following relations.

1.  $e_1 \prec e_2$  if  $\forall v. v(e_1) < v(e_2)$ , and
2.  $e_1 \preceq e_2$  if  $\forall v. v(e_1) \leq v(e_2)$ , and
3.  $e_1 \sim e_2$  if  $\forall v. v(e_1) = v(e_2)$ .

A level expression is in *normal form* if it is a level expression constant or  $k|i_1 n_1 \cdots i_r n_r$  for some distinct level variables  $i_j$  and some natural numbers  $n_j$ . We denote the normal forms of  $L$  as the set  $\|L\|$ .

**Lemma B.0.1** Every expression  $e$  has a normal form, *i.e.* there is an  $e' \in \|L\|$  such that  $e \sim e'$ .

**Proof:** We prove this by structural induction over  $e$ . The proof is trivial if  $e$  is a level constant.

Suppose  $e$  is a level variable. Let  $e' = [1|e0]$ . Then  $e' \in \|L\|$ . From the definition of level assignment,  $v(e) \geq 1$  for any level assignment. The fore for any level assignment  $v$ ,  $v(e') = \max(1, v(e)) = v(e)$ .

Suppose  $e = nk$ . Let  $e' = [1|nk]$ . Then  $e' \in \|L\|$ , and for any level assignment  $v$ ,  $v(e') = \max(1, v(n) + k) = v(n) + k$ .

Suppose  $e = e_1 | \cdots | e_r$ . By the induction hypothesis, each  $e_i$  has a normal form,  $e'_i$ . Suppose that for each  $e'_i$  that  $e'_i = c_i | n'_{i,1} k'_{i,1} \cdots n'_{i,s_i} k'_{i,s_i}$ . Let  $c = \max(c_1, \dots, c_r)$ . Let  $\{n_1, \dots, n_{r'}\}$  be the set of all of the level variables occurring in all of the  $e'_i$ 's.

For each  $n_i$ , let  $k_i \in \mathcal{N}$ , such that  $n_i k_i$  occurs in some  $e'_j$  and for all  $n'_i k$  that occur in  $e'_1, \dots, e'_r$   $k_i = \max(k_i, k)$ . Then for some arbitrary level assignment  $v$ ,

$$\begin{aligned}
v(e) &= \max(v(e_1), \dots, v(e_r)) \\
&= \max(v(e'_1), \dots, v(e'_r)) \\
&= \max(c_1, \dots, c_r, v(n'_{1,1} k'_{1,1}), \dots, v(n'_{r,s_r} k'_{i,s_r})) \\
&= \max(c, v(n_1 k_1), \dots, v(n_{r'} k_{r'})) \\
&= v(e').
\end{aligned}$$

□

Following from the definition of  $\sim$  and level variable assignment, if  $d \sim e$ ,  $d \sim e'$ , and  $e, e' \in \|L\|$  then only the constant levels occurring in  $e$  and  $e'$  can differ. For any  $d \in L$ , we define  $\|d\| = e \in \|L\|$  such that  $d \sim e$ , and for all  $e' \in \|L\|$  such that  $d \sim e'$ , the level constant expression occurring in  $e$  is less than or equal to level constant occurring in  $e'$ . We define  $\|\cdot\| \subseteq L \times \|L\|$  such that  $\langle e, e' \rangle \in \|\cdot\|$  if and only if  $\|e\| = e'$ .

We define  $\phi_v$ ,  $\phi_c$ ,  $\phi_i$ , and  $\phi_m$  to be predicates that are true for level expressions that are variables, constants, increments, and maximums, We define the substitution operation  $\sigma \subseteq L \times L$  as

$$\{\langle e, e' \rangle \mid v(e) = e' \text{ for some level assignment } v\}.$$

The set of predicates, operations, and relations over  $L$ ,  $L_O$ , is

$$\{\|\cdot\|, \phi_v, \phi_c, \phi_i, \phi_m, \sigma, \prec, \preceq, \sim\},$$

and the equivalence relation  $L_{\approx}$  is  $\sim$ .

## B.1 Computing the relations $\prec$ , $\preceq$ , and $\sim$

Given a level expression  $l = [k|v_1 i_1| \cdots |v_n i_n]$  in normal form, we define the floor of  $l$ , written  $\lfloor l \rfloor$ , as  $\max(k, 1 + i_1, \dots, 1 + i_n)$ . The floor of a level expression is the least value a level expression can take on under any level assignment.

**Lemma B.1.1** Given a level expression  $l$  in normal form,  $\lfloor l \rfloor \leq v(l)$  for all level assignments  $v$ .

**Proof:** The lemma follows from the fact that the least level constant that can be assigned to a level variable is 1.

□

Let  $l = [k|v_1 i_1| \cdots |v_n i_n]$  and  $l' = [k'|v'_1 i'_1| \cdots |v'_{n'} i'_{n'}]$  be level expressions in normal form. We define the relations  $\sqsubset$  and  $\sqsubseteq$  over level expressions in normal form as follows.

1.  $l \sqsubset l'$  if and only if  $\lfloor l \rfloor < \lfloor l' \rfloor$  and  $\forall j \in \{1|n\}$  exists  $j' \in \{1|n'\}$  such that  $v_j = v'_{j'}$  then  $i_j < i'_{j'}$ .
2.  $l \sqsubseteq l'$  if and only if  $\lfloor l \rfloor \leq \lfloor l' \rfloor$  and  $\forall j \in \{1|n\}$   $\exists j' \in \{1|n'\}$  such that  $v_j = v'_{j'}$  then  $i_j \leq i'_{j'}$ .

We can decide the relations  $\prec$ ,  $\preceq$ , and  $\sim$  by computing normal forms and applying the following lemma.

**Lemma B.1.2** Let  $e = c|n_1 k_1 \cdots n_r k_r$  and  $e' = c'|n'_1 k'_1 \cdots n'_{r'} k'_{r'}$ .

1. If  $e \sqsubset e'$  then  $e \prec e'$ .
2. If  $e \sqsubseteq e'$  then  $e \preceq e'$ .

3. If  $e \sqsubseteq e'$  and  $e' \sqsubseteq e$  then  $e \sim e'$ .

**Proof:** The proofs for (1) and (2) are the same, so we only prove (1) and (3). Suppose  $e \sqsubset e'$ . Let  $v$  be an arbitrary level assignment. From Lemma B.1.1,  $\lfloor e' \rfloor \leq v(e')$ . Since  $c \leq \lfloor e \rfloor < \lfloor e' \rfloor$ ,  $c < v(e')$ . Without loss of generality, assume that  $n_i = n'_i$  for  $i = 1, \dots, r$ . Since  $e \sqsubset e'$ ,  $k_i < k'_i$ . Therefore  $v(n_i k_i) < v(n'_i k'_i)$ . It follows that  $e \prec e'$ .

Suppose  $e \sqsubseteq e'$  and  $e' \sqsubseteq e$ . From the definition of  $\sqsubseteq$  we can conclude that  $r = r'$ . Suppose without loss of generality that  $n_i = n'_i$  for  $i = 1, \dots, r$ . Since  $e \sqsubseteq e'$  and  $e' \sqsubseteq e$ ,  $k_i = k'_i$ . Let  $v$  be an arbitrary level assignment, and let  $m = \max(v(n_1 k_1), \dots, v(n_r k_r))$ . Then  $v(e) = \max(c, m)$  and  $v(e') = \max(c', m)$ . We claim that if  $c \neq c'$  then  $c' \leq m$  and  $c \leq m$ . Suppose  $c \neq c'$ . Then  $\lfloor e \rfloor \neq c$  and  $\lfloor e' \rfloor \neq c'$ . Thus there exists an  $i$  such that  $c < k_i + 1$  and  $c < k'_i + 1$ . Since  $k_i + 1 \leq v(n_i k_i)$ ,  $k'_i + 1 \leq v(n'_i k'_i)$ , and  $v(n_i k_i) = v(n'_i k'_i)$ ,  $c' \leq m$  and  $c \leq m$ . Therefore if  $c \neq c'$  then  $\max(c, m) = m = \max(c', m)$ . Hence we can conclude that  $e \sim e'$ .

□

# Appendix C

## Refiner Interface

```
signature REFINER =
  sig
    type arg
    type argument
    type assumption
    eqtype tok = string
    type level_expression
    type parameter
    type proof
    type proof_annotation
    type rule
    type rule_term_tree
    type term
    datatype side_cond_val =
      SubgoalsSideCondVal of (assumption list * term) list
    | TermSideCondVal of term
    type tactic = proof -> (proof list * (proof list -> proof))
    eqtype variable

    val add_arg_stack_to_proof_annotation :
      (tok * arg) list list -> proof_annotation
      -> proof_annotation
    val add_full_label_to_proof_annotation :
      tok * (int,unit) sum -> proof_annotation
      -> proof_annotation
    val alpha_eq : term -> term -> bool
```

```

val annotate_proof : proof -> proof_annotation -> proof
val annotation_of_proof : proof -> proof_annotation
val arg_to_int : arg -> int
val arg_to_sub : arg -> (variable * term) list
val arg_to_tactic : arg -> tactic
val arg_to_term : arg -> term
val arg_to_tok : arg -> tok
val arg_to_var : arg -> variable
val argument_to_string : argument -> string
val arity_of_term : term -> int list
val bimodal_first_order_substitute :
  'a -> term -> (variable * term) list -> term
val bimodal_second_order_substitute :
  'a -> (tok * parameter) list
  -> (variable * (variable list * term)) list
  -> term -> term
val bindings_of_subterm_of_term : term -> int -> variable list
val bound_terms_of_term : term -> (variable list * term) list
val children : proof -> proof list
val clear_annotation_of_proof : proof -> proof
val compare_terms : term -> term -> bool
val compute : term -> term
val computeSubterms : term -> term
val conclusion : proof -> term
val constant_level_expression_p : level_expression -> bool
val construct_rule_term_tree :
  rule -> rule_term_tree list -> rule_term_tree
val create_reduction_rule : term -> unit
val debug_info : string -> unit
val debug_ref : bool ref
val define_new_rule : term -> unit
val delete_side_condition : string -> unit
val descriptor_of_slot_parameter : parameter -> tok
val destNatParm : parameter -> int
val destruct_assumption : assumption -> variable * (term * bool)
val destruct_assumption_list_argument : argument -> assumption list
val destruct_bool_parameter : parameter -> bool
val destruct_bound_term_argument : argument -> variable list * term
val destruct_constant_level_expression : level_expression -> int
val destruct_increment_level_expression : level_expression
  -> level_expression * int
val destruct_int_argument : argument -> int
val destruct_level_expression_argument :

```

```

        argument -> level_expression
val destruct_level_expression_parameter :
    parameter -> level_expression
val destruct_max_level_expression : level_expression
    -> level_expression list
val destruct_meta_parameter : parameter -> variable
val destruct_natural_parameter : parameter -> int
val destruct_parameter_argument : argument -> parameter
val destruct_parameter_substitution_list_argument :
    argument -> parameter list
val destruct_parameter_variable : parameter -> tok
val destruct_rule_term_tree : rule_term_tree
    -> rule * rule_term_tree list
val destruct_string_parameter : parameter -> string
val destruct_term : term
    -> (tok * parameter list)
        * (variable list * term) list
val destruct_term_argument : argument -> term
val destruct_term_using_parameter_bindings :
    term -> (tok * parameter list) * (parameter list * term) list
val destruct_token_argument : argument -> tok
val destruct_token_parameter : parameter -> tok
val destruct_variable_argument : argument -> variable
val destruct_variable_level_expression : level_expression -> tok
val destruct_variable_parameter : parameter -> variable
val display_message : string -> unit
val do_indicated_computations : term -> term
val edit_type_of_parameter : parameter -> tok
val equal_less_level_expression :
    level_expression -> level_expression -> bool
val equal_level_expression : level_expression -> level_expression
    -> bool

val equal_sequents : proof -> proof -> bool
val extract : proof -> unit
val extract_of_thm : tok -> term
val fail : string -> 'a
val first_order_match : term -> term
    -> variable list
    -> (variable * term) list

val free_variables : term -> variable list
val frontier : proof -> proof list * (proof list -> proof)
val get_arg_stack_of_proof_annotation :
    proof_annotation -> (tok * arg) list list

```

```

val get_full_label_of_proof_annotation : proof_annotation
                                     -> tok * (int,unit) sum

val stmt_of_thm : tok -> term
val hypotheses : proof -> assumption list
val hypotheses_equal : assumption list -> assumption list -> bool
val assumption_eq : assumption -> assumption -> bool
val id_of_rule : rule -> tok
val increment_level_expression_p : level_expression -> bool
val install_meta_eval_fun : string * (term list -> term) -> unit
val install_side_condition :
    string -> (assumption list * term
              -> argument list -> side_cond_val list)
              -> unit
val int_to_arg : int -> arg
val less_level_expression :
    level_expression -> level_expression -> bool
val level_variables_of_term : term -> tok list
val make_abstraction_meta_variable : tok -> variable
val make_assumption : variable -> term -> bool -> assumption
val make_assumption_list_argument : assumption list -> argument
val make_bool_parameter : bool -> parameter
val make_bound_term_argument : variable list -> term -> argument
val make_constant_level_expression : int -> level_expression
val make_display_meta_variable : tok -> variable
val make_edit_parameter : tok -> tok -> tok -> string -> parameter
val make_increment_level_expression : level_expression
                                     -> int -> level_expression
val make_int_argument : int -> argument
val make_level_expression_argument : level_expression -> argument
val make_level_expression_parameter : level_expression -> parameter
val make_max_level_expression :
    level_expression list -> level_expression
val make_meta_parameter : tok -> variable -> parameter
val make_natural_parameter : int -> parameter
val make_parameter_argument : parameter -> argument
val make_parameter_substitution_list_argument :
    parameter list -> argument
val make_parameter_variable : tok -> tok -> parameter
val make_primitive_rule : tok -> argument list -> rule
val make_proof_node : assumption list -> term -> proof
val make_string_parameter : string -> parameter
val make_tactic_rule : term -> rule
val make_term : tok * parameter list

```

```

-> (variable list * term) list -> term
val make_term_argument : term -> argument
val make_term_using_parameter_bindings :
  tok * parameter list -> (parameter list * term) list
  -> term
val make_tok_argument : tok -> argument
val make_token_parameter : tok -> parameter
val make_variable_argument : variable -> argument
val make_variable_level_expression : tok -> level_expression
val make_variable_parameter : variable -> parameter
val max_level_expression_p : level_expression -> bool
val meta_parameter_p : parameter -> bool
val meta_type_of_parameter : parameter -> tok
val meta_variable_type : variable -> tok
val mkFoSubst : ('a * term) list
  -> ('a * term * variable list * 'b list) list
val most___recently___used___tactic: tactic ref

val name_of_meta_parameter : parameter -> tok
val natural_parameter_of_term : term -> int -> int
val normalize_level_expression :
  level_expression -> level_expression
val null_proof_annotation : proof_annotation
val operator_id_of_term : term -> tok
val operator_of_term : term -> tok * parameter list
val parameter_eq : parameter -> parameter -> bool
val parameter_of_term : term -> int -> parameter
val parameter_to_string : parameter -> string
val parameters_of_term : term -> parameter list
val prlgoal : proof
val proof_to_string : proof -> string
val refine : rule -> proof ->
  proof list * (proof list -> proof)
val refinement : proof -> rule
val rule_to_term : rule -> term
val second_order_free_variables : term -> (variable * int) list
val setAnnotation : proof -> proof_annotation option -> proof
val string_of_error_parameter : parameter -> string
val string_of_parameter : parameter -> string
val sub_to_arg : (variable * term) list -> arg
val substitute_in_level_expression :
  level_expression -> (tok * level_expression) list
  -> level_expression

```

```

val subterm_of_term : term -> int -> term
val tactic_to_arg : tactic -> arg
val term_to_arg : term -> arg
val term_to_rule : term -> rule
val term_to_string : term -> string
val tok_to_arg : tok -> arg
val tok_to_variable : tok -> variable
val token_parameter_of_term : term -> int -> tok
val type_of_argument : argument -> tok
val type_of_parameter : parameter -> tok
val var_to_arg : variable -> arg
val variable_level_expression_p : level_expression -> bool
val variable_parameter_of_term : term -> int -> variable
val variable_to_tok : variable -> tok
val insert_rewrite_rule : term * term -> unit
val delete_rewrite_rule : term -> unit
val name_of_matching_rewrite_rule : term -> string option
val insert_thm : string -> proof -> unit
val delete_thm : string -> unit
val proof_finished_p : proof -> bool

structure RefinementServer :
  sig
    eqtype req_id
    eqtype server_id
    type reply
    type server
    type request_handler

    val enqRequest : (tactic * proof) -> req_id

    val enqRequests : (tactic * proof) list -> req_id list

    val deqRequest : unit -> (req_id * (tactic * proof)) option

    val deqRequests : int -> (req_id * (tactic * proof)) list

    val insertReply : req_id * reply -> unit

    val getReply : req_id -> reply option
  end

```

```
val waitForReply : req_id -> request_handler -> reply

val allReplies : req_id list -> request_handler -> reply list

val discardReplies : req_id list -> unit

val server : int -> ((tactic * proof)
                    -> (proof list * (proof list -> proof)))
                    -> server

val spawn: server -> server_id option

val processReply : reply -> (proof list * (proof list -> proof))

val requestHandler : ((tactic * proof) ->
                     (proof list * (proof list -> proof)))
                     -> request_handler

val serviceRequest : request_handler -> (tactic * proof)
                    -> reply

val fork : (unit -> unit) -> server_id option

end

end
```

# Appendix D

## Refinement Rule Specification

# Syntax

```

⟨rule-spec⟩      : !rule_specification(⟨sequent⟩; ⟨rule⟩;
                    ⟨let⟩ let_cons list; ⟨subgoals⟩)
⟨sequent⟩       : !sequent(⟨assumption⟩ !assum_cons list; ⟨term⟩; ⟨term⟩)
⟨rule⟩          : !rule{⟨tok⟩:t}(⟨rule-arg⟩ !rule_arg_cons list)
⟨assumption⟩    : ⟨assumption-list⟩
                  — !assumption{⟨hidden⟩:b,⟨id⟩:v}(⟨type⟩)
⟨assumption-list⟩ : !assumption_list{⟨tok⟩:t}
                  — !subst(⟨assumption-list⟩;⟨substitution-list⟩)
⟨rule-arg⟩      : !term(⟨term⟩)
                  — !assumption_index{⟨tok⟩:t}
                  — !variable{⟨id⟩:v}
                  — !parameter{⟨id⟩:⟨type⟩}
                  — !bound_id(⟨id⟩ list.⟨term⟩)
                  — !substitution_list{⟨tok⟩:t}
                  — !assumption_list{⟨tok⟩:t}
                  — ⟨subst⟩

```

```

⟨let⟩           : !let(⟨term⟩; ⟨term⟩)
                — !let(⟨let-lhs-arg⟩ !rule_arg_cons list;
                    !call_lisp{⟨tok⟩:t})

⟨let-lhs-arg⟩   : ⟨rule-arg⟩
                — !goal_list{⟨tok⟩:t}

⟨subgoals⟩     : ⟨sequent⟩ !goal_cons list
                — !goal_list{⟨tok⟩:t}

⟨term⟩         : ⟨subst⟩
                — !variable{⟨id⟩:v}
                — term

⟨subst⟩        : !subst(⟨term⟩; ⟨substitution-list⟩)

⟨substitution-list⟩ : !substitution_list{⟨tok⟩:t}
                — ⟨sub⟩ !sub_cons list

⟨sub⟩         : !term_sub{⟨v⟩:v}(⟨term⟩)
                — !parameter_sub{⟨le-var⟩:l, ⟨le-value⟩:l}
                — !parameter_sub{⟨$id⟩:⟨type⟩, ⟨value⟩:⟨type⟩}

```

# Appendix E

## Preliminary Tests of the MP Refiner

Table E.1, Table E.2, and Table E.3 contain the results of our preliminary tests of AND-parallelism with the MP refiner, the speedups of tactics created with `PTHEN`. The tactics in the column labeled **Lhs** are the tactics used on the left hand side of `PTHEN` and the tactics in the column labeled **Rhs** are the tactics used on the right hand side of `PTHEN`. The speedups were obtained using the MP refiner using two, three, and four refinement servers.

Table E.4, Table E.5, Table E.6, and Table E.7 contain results of our preliminary test of OR-parallelism with the MP refiner, the speedups of tactics created with `PORELEL`. In Table E.4 and Table E.5 the first tactic succeeds. In Table E.6 and Table E.7 only the last alternative succeeds. The alternatives are the tactics in the column labeled **Rhs**. The entries in the column labeled **alts** indicates the

number of alternatives for each tactic. The speedups were obtained using the MP refiner using two, three, and four refinement servers.

Table E.1: Preliminary Results of AND-parallelism (Part 1)

<b>Lhs</b>	<b>Rhs</b>	<b>p = 2</b>	<b>p = 3</b>	<b>p = 4</b>
tac1_1	tac1_1	0.7640	0.7070	0.5260
tac1_1	tac65_10	0.6450	0.5020	0.3950
tac1_1	tac400_100	0.8550	0.7600	0.6540
tac1_1	tac1000_300	0.9620	0.8980	0.7540
tac1_1	tac2000_500	0.6980	0.6440	0.5790
tac1_1	tac5000_100	0.7130	0.6700	0.5780
tac1_1	tac10000_100	0.7550	0.7130	0.5680
tac1_2	tac1_1	0.8990	0.8020	0.6300
tac1_2	tac65_10	1.2470	0.9370	0.6820
tac1_2	tac400_100	1.6610	1.5320	1.3890
tac1_2	tac1000_300	1.7180	1.4790	0.0000
tac1_2	tac2000_500	1.2820	1.2880	1.1610
tac1_2	tac5000_100	1.2890	1.2760	1.0780
tac1_2	tac10000_100	1.3120	1.2820	1.0160

Table E.2: Preliminary Results of AND-parallelism (Part 2)

<b>Lhs</b>	<b>Rhs</b>	<b>p = 2</b>	<b>p = 3</b>	<b>p = 4</b>
tac1_4	tac1_1	1.0770	1.0170	0.8010
tac1_4	tac65_10	1.3620	1.1530	1.2070
tac1_4	tac400_100	1.7340	1.6500	2.4010
tac1_4	tac1000_300	1.7620	1.5740	2.2420
tac1_4	tac2000_500	1.3340	1.2620	1.2730
tac1_4	tac5000_100	1.2590	1.2270	1.6280
tac1_4	tac10000_100	1.2700	1.2270	1.5960
tac1_10	tac1_1	0.6730	1.2740	1.1070
tac1_10	tac65_10	1.5890	1.5740	1.6450
tac1_10	tac400_100	1.7320	1.8830	2.2500
tac1_10	tac1000_300	1.7270	1.9160	2.3020
tac1_10	tac2000_500	1.2850	1.4560	1.5130
tac1_10	tac5000_100	1.2510	1.4740	1.6100
tac1_10	tac10000_100	1.2730	1.5390	1.3830
tac1_20	tac65_10	1.7390	1.9410	2.0120
tac1_30	tac65_10	1.4280	2.0080	1.7280

Table E.3: Preliminary Results of AND-parallelism (Part 3)

<b>Lhs</b>	<b>Rhs</b>	<b>p = 2</b>	<b>p = 3</b>	<b>p = 4</b>
tac1_50	tac1_1	1.2120	1.0660	1.0180
tac1_50	tac65_10	1.7730	2.2410	2.5580
tac1_50	tac400_100	1.5840	1.8880	2.3610
tac1_50	tac1000_300	1.5290	1.8930	2.2350
tac1_50	tac2000_500	1.2180	1.5880	0.0000
tac1_50	tac5000_100	1.2640	1.5710	0.0000
tac1_50	tac10000_100	1.1570	1.5510	1.6170
tac1_100	tac1_1	1.0300	0.9870	0.8930
tac1_100	tac65_10	1.7910	2.3390	2.8250
tac1_100	tac400_100	1.6060	1.9320	2.2300
tac1_100	tac1000_300	1.5640	1.8720	2.2880
tac1_100	tac2000_500	1.2460	1.5040	1.7400
tac1_100	tac5000_100	1.2200	1.5420	1.6610
tac1_100	tac10000_100	1.1700	1.4130	1.5510

Table E.4: Preliminary Results of OR-Parallelism (Part 1)

Lhs	Rhs	alts	p = 2	p = 3	p = 4
tac65_10	tac1_1	1	0.8760	0.6460	0.4290
tac65_10	tac65_10	1	0.6340	0.4820	0.4450
tac65_10	tac400_100	1	0.4400	0.3610	0.4040
tac65_10	tac1000_300	1	0.4500	0.3730	0.3690
tac65_10	tac2000_500	1	0.3840	0.3180	0.3070
tac65_10	tac5000_100	1	0.3830	0.3200	0.2770
tac65_10	tac10000_100	1	0.3860	0.3290	0.2780
tac65_10	tac1_1	5	0.8330	0.6210	0.4550
tac65_10	tac65_10	5	0.5030	0.3730	0.3030
tac65_10	tac400_100	5	0.4810	0.3840	0.3010
tac65_10	tac1000_300	5	0.4970	0.3670	0.3000
tac65_10	tac2000_500	5	0.3890	0.3050	0.2490
tac65_10	tac5000_100	5	0.3830	0.3110	0.2690
tac65_10	tac10000_100	5	0.3940	0.3200	0.2450

Table E.5: Preliminary Results of OR-Parallelism (Part 2)

Lhs	Rhs	alts	p = 2	p = 3	p = 4
tac65_10	tac1_1	20	0.8580	0.6450	0.4940
tac65_10	tac65_10	20	0.5410	0.3850	0.3230
tac65_10	tac400_100	20	0.5340	0.3990	0.3050
tac65_10	tac1000_300	20	0.5320	0.3880	0.3170
tac65_10	tac2000_500	20	0.4110	0.3370	0.2480
tac65_10	tac5000_100	20	0.4300	0.3320	0.2490
tac65_10	tac10000_100	20	0.4240	0.3320	0.2650
tac65_10	tac1_1	50	0.8590	0.6510	0.5080
tac65_10	tac65_10	50	0.6010	0.3660	0.2990
tac65_10	tac400_100	50	0.5680	0.3680	0.2970
tac65_10	tac1000_300	50	0.5970	0.3660	0.2860
tac65_10	tac2000_500	50	0.4170	0.3390	0.2370
tac65_10	tac5000_100	50	0.4250	0.3210	0.2600
tac65_10	tac10000_100	50	0.4220	0.3240	0.2520

Table E.6: Preliminary Results of OR-Parallelism (Part 3)

Lhs	Rhs	alts	p = 2	p = 3	p = 4
tac65_10	tac1_1	1	0.8550	0.6460	0.4220
tac65_10	tac65_10	1	1.2220	0.9590	0.8620
tac65_10	tac400_100	1	0.9870	0.8710	0.7970
tac65_10	tac1000_300	1	1.0040	0.9390	0.7870
tac65_10	tac2000_500	1	0.7470	0.6260	0.3130
tac65_10	tac5000_100	1	0.7190	0.6190	0.5770
tac65_10	tac10000_100	1	0.7690	0.6840	0.6070
tac65_10	tac65_10	3	1.5740	1.3010	1.4840
tac65_10	tac65_10	4	1.3810	1.6060	1.2310
tac65_10	tac1_1	5	0.8880	0.6560	0.4800
tac65_10	tac65_10	5	1.6550	1.8210	1.4780
tac65_10	tac400_100	5	1.5360	1.9850	1.7510
tac65_10	tac1000_300	5	1.5430	1.9120	1.7190
tac65_10	tac2000_500	5	1.1080	1.3630	0.8600
tac65_10	tac5000_100	5	1.1070	1.3430	1.1760
tac65_10	tac10000_100	5	1.0520	1.4050	1.2050

Table E.7: Preliminary Results of OR-Parallelism (Part 4)

Lhs	Rhs	alts	p = 2	p = 3	p = 4
tac65_10	tac1_1	20	1.0420	0.7860	0.6080
tac65_10	tac65_10	20	1.7210	2.2650	2.3660
tac65_10	tac400_100	20	1.7220	2.1940	2.6820
tac65_10	tac1000_300	20	1.7530	2.1620	2.6820
tac65_10	tac2000_500	20	1.2910	1.4430	1.9170
tac65_10	tac5000_100	20	1.2810	1.5330	1.7360
tac65_10	tac10000_100	20	1.2460	1.5030	1.7150
tac65_10	tac1_1	50	1.3600	1.0010	0.7530
tac65_10	tac65_10	50	1.7680	2.3560	2.7740
tac65_10	tac400_100	50	1.7320	2.2000	2.7290
tac65_10	tac1000_300	50	1.7120	2.2140	2.6680
tac65_10	tac2000_500	50	1.3100	1.6010	2.0280
tac65_10	tac5000_100	50	1.2670	1.6470	1.8430
tac65_10	tac10000_100	50	1.2990	1.2320	1.8300

# Bibliography

- [1] M.D. Aagaard and M.E. Leeser. Verifying a logic synthesis tool in nuprl. In *Computer-Aided Verification, CAV '92*, number 663 in *Lecture Notes in Computer Science*, pages 77–83. Springer-Verlag, 1992.
- [2] M.D. Aagaard, M.E. Leeser, and P.J. Windley. Toward a super duper hardware tactic. In *Higher Order Theorem Proving and its Applications*, number 780 in *Lecture Notes in Computer Science*, pages 400–412. Springer-Verlag, 1993.
- [3] William E. Aitken. *Reflecting on Nuprl*. Ph.D. dissertation, Cornell University, forthcoming.
- [4] S.F. Allen, R.L. Constable, and D.J. Howe. Reflecting the open-ended computation system of constructive type theory. In F.L. Bauer, editor, *Logic, Algebra and Computation*, pages 267–288. Springer-Verlag, 1991.
- [5] S.F. Allen, R.L. Constable, D.J. Howe, and W.E. Aitken. The semantics of reflected proof. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105. IEEE Computer Society Press, 1990.
- [6] Stuart F. Allen. *A Non-type-theoretic Semantics for a Type Theoretic Language*. Ph.D. dissertation, Cornell University, 1987.
- [7] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software, Practice, and Experience*, 19:275–279, 1989.
- [8] Andrew W. Appel. A runtime system. *Journal of Lisp and Symbolic Computation*, 3, 1990.
- [9] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

- [10] Andrew W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3:391–430, 1993.
- [11] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming: 3rd International Symposium*, pages 1–13. Springer-Verlag, 1991.
- [12] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 13–23. ACM Press, 1994.
- [13] Joseph Bates and Robert Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7:113–136, 1985.
- [14] Joseph L. Bates. *A Logic for Correct Program Development*. Ph.D. dissertation, Cornell University, 1979.
- [15] Nikolaj Bjorner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomas E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Computer Aided Verification, CAV '96*, number 1101 in *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, 1996.
- [16] Maria Paola Bonacina and Jieh Hsiang. Distributed deduction by clause-diffusion. Technical report, Department of Computer Science SUNY at Stony Brook, 1992.
- [17] Soumitra Bose et al. Parthenon: A parallel theorem prover for non-horn clauses. *Journal of Automated Reasoning*, 8:153–181, 1989.
- [18] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [19] Thomas F. Coleman, David Shalloway, and Zhijun Wu. A parallel build-up algorithm for global energy minimizations of molecular clusters using effective energy simulated annealing. Technical Report CTC93TR13, Cornell Theory Center, 1993.
- [20] R. L. Constable, T. B. Knoblock, and J. L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1:285–326, 1985.
- [21] Robert Constable. The structure of Nuprl's type theory. In Helmut Schwichtenberg, editor, *Logic of Computation*, pages 123–145. Springer, Berlin, 1997.

- [22] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986.
- [23] Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*. Number 135 in *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1982.
- [24] C. Cornes, J. Courant, J. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. *The Coq Proof Assistant Reference Manual: Version 5.10*. Unpublished, 1995.
- [25] N.J. deBruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, number 125 in *Lecture Notes In Mathematics*, pages 29–61. Springer Verlag, 1968.
- [26] John December. *Presenting Java: An Introduction to the Java Language and HotJava Browser*. SAMS.Net, Indianapolis, Indiana, 1995.
- [27] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: A system description. In *Eleventh Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1990.
- [28] L.M.G. Feijs, H.B.M. Jonkers, and C.A. MiddelBurg. *Notations for Software Design*. Springer-Verlag, London, 1994.
- [29] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufman, San Francisco, 1994.
- [30] M. J. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
- [31] Michael Gordon, Arthur Milner, and Christopher Wadsworth. *Edinburgh LCF*. Number 78 in *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.
- [32] Timothy G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*. Ph.D. dissertation, Cornell University, 1989.
- [33] R. H Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transaction of Programming Languages and Systems*, 7:501–538, 1985.

- [34] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. In *Record of the ACM-SIGPLAN Workshop on ML and its Applications*, number 2265 in INRIA Research Report, pages 136–147, 1994.
- [35] Jason Hickey. Nuprl-Light: An implementation framework for higher order logics. In *Automated Deduction, CADE 14*, number 1249 in *Lecture Notes in Artificial Intelligence*, pages 395–399. Springer-Verlag, 1997.
- [36] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- [37] Gerard Huet. A uniform approach to type theory. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, pages 337–398. Addison-Wesley, 1990.
- [38] Gerard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
- [39] Paul Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. Ph.D. dissertation, Cornell University, 1995.
- [40] Matt Kaufmann. A user’s manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 19, Computational Logic, Inc., 1988.
- [41] Fredrick Colvin Knabe. *Language Support For Mobile Agents*. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1995.
- [42] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.
- [43] Vipin Kumar et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.
- [44] Xavier Leroy. The Objective Caml system, documentation, and user’s guide. Unpublished, 1997.
- [45] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A high performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

- [46] Ewing L. Lusk and William W. McCune. Experiments with Roo, a parallel automated deduction system. In *Parallelization in Inference Systems*, number 590 in *Lecture Notes in Artificial Intelligence*, pages 139–162. Springer-Verlag, 1990.
- [47] W. W. McCune. OTTER 1.0 user’s guide. Technical Report ANL-88-44, Argonne National Laboratory, 1989.
- [48] Robin Milner, Mads Tofte, and Robert Harper. *The Definition Of Standard ML*. MIT Press, Cambridge, MA, 1990.
- [49] Dan I. Moldvan. *Parallel Processing: From Applications to Systems*. Morgan Kaufman, San Mateo, CA, 1993.
- [50] J. Gregory Morrisett and Andrew Tolmach. A portable multiprocessor interface for Standard ML of New Jersey. Technical Report CMU-CS-92-155, School of Computer Science Carnegie Mellon University, 1992.
- [51] Roderick Moten. Nuprl as a generic theorem prover. Technical Report TR96-1578, Cornell University, 1996.
- [52] Ake Nordlund, Klaus Galsgaard, and R. F. Stein. Visualizing astrophysical 3d MHD turbulence. In *Applied Parallel Computing: Computations in Physics, Chemistry, and Engineering Science*, number 1041 in *Lecture Notes In Computer Science*, pages 450–461. Springer Verlag, 1996.
- [53] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Eleventh International Conference on Automated Deduction (CADE)*, number 607 in *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, 1992.
- [54] Same Owre et al. Formal verification of fault tolerance architectures: A prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21:107–125, 1995.
- [55] Larry Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, Cambridge, 1987.
- [56] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.
- [57] Lawrence C. Paulson. *ML for the Working Programmer (Second Edition)*. Cambridge University Press, Cambridge, 1996.

- [58] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium 73*, number 80 in *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1973.
- [59] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology, and Philosophy of Science VI*, pages 153–175. North-Holland, 1982.
- [60] R. Pollack. Lego user’s guide. Technical report, University of Edinburgh, 1990.
- [61] John H. Reppy. Concurrent ML: Design, application and semantics. In ?, editor, *Programming, Concurrency, Simulation and Automated Reasoning*, number 693 in *Lecture Notes in Computer Science*, pages 165–198. Springer-Verlag, 1992.
- [62] John H. Reppy. *Higher-Order Concurrency*. Ph.D. dissertation, Cornell University, 1992.
- [63] John H. Reppy. A high performance garbage collector for Standard ML. Technical Report Memo, AT & T Bell Laboratories, 1993.
- [64] P. Rudnicki. An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs*. Bastad, 1992.
- [65] J. Schumann and R. Letz. Partheo: A high-performance parallel theorem prover. In *Tenth International Conference on Automated Deduction (CADE)*, number 449 in *Lecture Notes in Artificial Intelligence*, pages 40–56. Springer-Verlag, 1990.
- [66] Johann M. Ph. Schumann. Parallel theorem provers: An overview. In *Parallelization in Inference Systems*, number 590 in *Lecture Notes in Artificial Intelligence*, pages 26–50. Springer-Verlag, 1990.
- [67] John K. Slaney and Ewing L. Lusk. Parallelizing the closure computation in automated deduction. In *Tenth International Conference on Automated Deduction (CADE)*, number 449 in *Lecture Notes in Artificial Intelligence*, pages 28–39. Springer Verlag, 1990.
- [68] Christian B. Suttner. A parallel theorem prover with heuristic work distribution. In *Parallelization in Inference Systems*, number 590 in *Lecture Notes In Computer Science*, pages 243–253. Springer Verlag, 1992.

- [69] Christian B. Suttner. *Parallelization of Search-based Systems by Static Partitioning with Slackness*. Ph.D. dissertation, Institut für Informatik, TU München, 1995.
- [70] Christian B. Suttner and Manfred B Jobmann. Simulation analysis of static partitioning with slackness. In *Parallel Processing for Artificial Intelligence 2*, number 15 in Machine Intelligence and Pattern Recognition, pages 93–105. North-Holland, 1994.
- [71] Christian B. Suttner and Johann Schumann. Parallel automated theorem proving. In *Parallel Processing for Artificial Intelligence 1*, number 14 in Machine Intelligence and Pattern Recognition, pages 209–257. North-Holland, 1994.
- [72] Simon Thompson. *Type Theory and Functional Programming*. Addison Wesley, Wokingham, England, 1991.
- [73] M. Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.
- [74] D. H. D. Warren. The SRI model for OR-parallel execution of Prolog: Abstract design and implementation issues. In *International Symposium on Logic Programming*, pages 92–102. North-Holland, 1987.
- [75] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suarez. The CAML reference manual. Technical Report 121, INRIA, 1991.
- [76] G.A. Wilson and J. Minker. Resolution, refinements, and search strategies: A comparative study. *IEEE Transactions on Computers*, C-25:782–801, 1976.