Type-Theoretic Models of Concurrency

Walter Rance Cleaveland II

Ph. D. Thesis

87-837
May 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

TYPE-THEORETIC MODELS OF CONCURRENCY

A Thesis

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

Walter Rance Cleaveland II

May 1987

TYPE-THEORETIC MODELS OF CONCURRENCY

Walter Rance Cleaveland II, Ph.D.
Cornell University, 1987

Sequential computation has well-understood correctness criteria and proof techniques for verifying programs, but the novelty and complexity of concurrent computation complicates a similar analysis of concurrent programs. This thesis examines the use of a system for developing formal mathematics, the Nuprl proof development system, as a tool for reasoning about concurrency and ameliorating somewhat the complex chore of analyzing concurrent programs.

Two approaches to using Nuprl in this fashion are presented. In the first, semantic models of concurrent computation are developed in the Nuprl formalism, and the Nuprl logic is used to reason about the properties of objects in the model and to develop other logics for reasoning about objects in the semantic model. These objects represent the meanings of programs, so properties of these objects will be properties of the programs they represent. Both models are used to give a semantic account of the program constructors of Milner's CCS and Hoare's CSP, and a development of temporal logic is given in the context of one model. An actual Nuprl implementation of the other model is described.

In the second approach, a means of introducing parallelism implicitly into the Nuprl evaluation framework is presented and studied. Nuprl embodies a programming language and supplies a mechanism for

conducting verified programming. Therefore, incorporating a parallel evaluation procedure for the programming language introduces a notion of verified concurrent execution into the system.

Biographical Sketch

Walter Rance Cleaveland II was born in Baltimore, Maryland on July 18, 1961. After brief stints in various elementary schools, where a fear of being whacked on the backside was instilled in the impressionable Mr. Cleaveland, he landed at the McCallie School for Boys, a preparatory school in Chattanooga, Tennessee, where the students were encouraged, in the words of the school motto, "to glorify God and to enjoy Him forever." In the course of doing so, Mr. Cleaveland stumbled across a textbook in Geometry that affected him profoundly; unfortunately, the book was wielded threateningly by an older boy, and it took Mr. Cleaveland several years to recover from this peculiar strain of "Math Anxiety." Happily, he did, and he went to Duke University in the fall of 1978, where, in 1982, he was graduated *summa cum laude* with a B.S. in Mathematics and Computer Science. In the fall of 1982 Mr. Cleaveland loaded his 1975 Plymouth Valiant and ventured across the frozen Northern tundra to pursue his doctoral work at Cornell University, where he would learn the meaning of the word "cold" and earn his M.S. in Computer Science in June of 1985. His life did warm up substantially in 1986, however, as he renewed the acquaintance of a young lady, Karen Hardee, who agreed, in 1987, to marry him in 1988. Mr. Cleaveland leaves Cornell to work at the University of Sussex in Brighton, England.

Dedication

To my parents, Clif and Ruzha, and to Karen

Acknowledgements

My adviser, Robert Constable, has played a major role in my thesis work; his insights into type theory were responsible for stimulating my interest in Nuprl and starting my research career. Every thesis writer has his mentor, and Prakash Panangaden has ably filled this capacity for me; any expression of gratitude to him almost seems inadequate for the encouragement and guidance he has given me. Abha Moitra served on my committee and helped me clarify certain points in my thesis, and Fred Schneider made useful remarks at early stages in my research. I am grateful to Keshav Pingali for providing me with the inspiration for the strictness analysis in chapter 7. I also wish to thank Doug Howe for the assistance he gave me as I struggled with Nuprl and Todd Knoblock for the improvements he suggested to the introduction.

The National Science Foundation, IBM and Cornell have supported me generously with fellowships during my stay at Cornell, and I am very grateful for their financial assistance.

Personally, my life at Cornell has been marked by ups and downs, but the ups would not have been so high, and the downs would have been a lot lower, without the support of my family and friends. Each in their own way has enriched my experience at Cornell immeasurably, and in this I feel blessed. And what should have been an awful time (the writing of my thesis) turned out to be a wonderful time, indeed, because of Karen Hardee; Ithaca is a great place to fall in love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel computation represents a potentially very successful alternative to the vonNeumann notion of sequential computation. The interest in parallelism has risen in response to several different needs in the computing community. On the one hand, many kinds of problems involving the management of distributed resources are best addressed in a parallel fashion; one naturally thinks in terms of processes governing each piece of the distributed resource and communicating with each other in order to maintain system constraints. On the other hand, as the physical limits of the speed of sequential computation are approached, additional acceleration of the speed of computation necessitates the examination of alternative paradigms for computation.

One major impediment to the realization of the potential of parallel computation, however, stems from the intellectual difficulty inherent in programming parallel computers. Implementing parallel computation, of course, requires the writing at some level of programs that execute in parallel, and determining whether these programs satisfy the properties that the designer intends is an extremely subtle task. In general there is a lack of agreed upon semantics for parallel computation, meaning that it is difficult to explain how concurrent programs work and consequently difficult to teach programmers how to write parallel programs. Also, the details of the concurrently executing pieces of each program must be followed simultaneously during the development of a parallel program, requiring a large degree of intellectual effort on the part program designers to understand their programs. The possibility of shared resources complicates the task of writing correct programs even futher. Formalisms have been proposed that enable properties of concurrent programs to be stated and proven correct, but these formalisms necessarily involve the tracking of a substantial amount of detail. Certainly, some degree of automated assistance would aid the process of verification immensely, and this thesis examines one way in which such automation might be accomplished.

The work in this thesis was motivated by our desire to use the Nuprl proof development system [Constable et al. 86] to reason about a formalism for concurrent computation, Milner's Calculus for Communicating Systems [Milner 80], and to examine how parallelism may be incorporated into the Nuprl programming language. Using Nuprl as a testbed for reasoning about CCS seems a natural approach to examining the issues surrounding the automation of the verification process. Also, since the Nuprl system provides a mechanism for programming in a verified fashion, examining ways to incorporate concurrency into Nuprl programming provides another means of verifying concurrent programs. It is the purpose of this thesis, then, to present mathematical models for concurrency, to examine their ability to model CCS and their suitability for automation in the Nuprl environment, and to examine the feasibility of incorporating parallelism directly into the Nuprl theory.

The appeal of using Nuprl as a metasystem for formal reasoning stems from the support the system provides for making formal mathematical manipulations feasible. Working within a formal mathematical system has advantages and drawbacks. The most important advantage is that developing mathematics in a formal fashion forces the mathematician to be rigorous, and the resulting work has a greater likelihood of being correct. This enforced rigor, however, also results in the biggest drawback to formal mathematics, which is the tedium inherent in addressing every detail in a mathematical argument. It is this tedium that Nuprl attempts to alleviate. In addition to embodying a foundational theory of mathematics, constructive type theory, this system incorporates a proof editor for aiding in the development of proofs and a very flexible means for encapsulating low-level notations into higher-level ones. The system also supplies a means of developing consistent proof rules on top of the rules given by the type theory, so that inference rules can be defined that closely correspond to the large steps of inference typically found in more informal mathematics. Furthermore, the system embodies a programming language based on the lambda calculus that allows computational information to be extracted from proofs of theorems.

The attractiveness of the CCS formalism lies in its abstractness and in the novelty of the style of reasoning proposed for it. The calculus is algebraic in the sense that an equational logic is associated with it; reasoning about objects defined using the term constructors in CCS can therefore be done algebraically. The calculus also dispenses with issues not relating directly to concurrency; thus, the concerns that one has in reasoning about CCS are concerns one will have in reasoning about any concurrent programming language.

1.1. The Semantic-Based Approach

One of the characteristics of concurrent program verification is the variety of styles of reasoning different researchers employ. Some of these different methods are surveyed in chapter 2. One of our goals is to develop a framework in which accounts of many of these methods may be given, and the approach we advocate for doing so is a *semantic-based* strategy. Essentially, this kind of approach requires that the semantics of the programming system be rigorously developed in some model, with reasoning then taking place about the objects in the model corresponding to programs. Using this approach it is possible to give formal accounts of both equational and logical styles of reasoning in the following sense. Reasoning about equivalence between programs involves reasoning about equality in the semantic model; reasoning about properties of programs involves proving theorems in a logic with the class of individuals taken to be the objects in the theory.

Other advantages to reasoning this way include the fact that consistency of logics for reasoning about concurrency can be assured and the fact that it is possible to develop logics for reasoning about concurrency independently of a particular programming language. Developing a new programming logic in a semantic-based framework requires that the semantics of the logic be specified in terms of the objects in the semantic model and a logic already developed for reasoning about objects in the semantic theory. Assuming the "base logic" associated with the semantic model is consistent (and in this thesis the base logic we work with is the predicate calculus implicit in Nuprl, which is known to be consistent), the newly developed logic must be consistent, since its semantic-based development essentially specifies a model for it. Also, by having logics refer to objects in the semantic model of concurrency rather than programs written in a particular language, the same logic may be conveniently used to reason about different programming languages, provided the languages may be given semantics in the model. Brookes [Brookes 85, Brookes 86] has also advocated semantic-based approaches to reasoning, although he favors axiomatic semantics as opposed to our more denotational approach.

That such a strategy is feasible denotationally was suggested to us by our experience with the Nuprl type theory. In addition to embodying the constructive predicate calculus, this theory also supplies a mechanism for reasoning about equality of objects within types. Therefore, identifying a type with a structure sufficiently rich to model process behavior enables both equational and propositional reasoning about properties of programs to be carried out.

1.2. Incorporating Parallelism into Nuprl

The constructive nature of the Nuprl logic allows proofs of theorems to be interpreted as programs. Under this interpretation statements of theorems are viewed as specifications, and the proofs of theorems are viewed as programs that meet the specifications implicit in the theorems. Since the Nuprl theory is sound, proofs are guaranteed to be valid, implying that the programs extracted from proofs are guaranteed to satisfy their specifications.

Nuprl programs are evaluated sequentially; one natural means, then, of introducing parallelism into Nuprl involves making the evaluation procedure parallel. Provided the new evaluation procedure obeys the semantics of Nuprl, the programs corresponding to proofs of theorems execute in parallel in a correct fashion. In a certain sense, then, using an appropriate parallel evaluation procedure would enable "parallel" Nuprl programs to be written that are guaranteed to be correct. We present such a procedure later in this thesis.

1.3. Organization of the Thesis

The rest of the thesis is organized along the following lines. Chapter 2 presents an overview of some of the literature about theoretical aspects of concurrency. Chapter 3 describes the Nuprl theory and system in more detail. Chapters 4 and 5 each present a different model of concurrency; the merits of each are discussed, and each model is developed in the Nuprl type theory. Chapter 6 describes an implementation of the model described in chapter 5 on the Nuprl system and a development of Milner's Calculus for Communicating Systems in the model. Chapter 7 describes how concurrent execution may be incorporated into the Nuprl theory itself, and chapter 8 presents our conclusions and suggests areas of further research.

Chapter 2

A Survey of Concurrency

A number of formalisms have been proposed for reasoning about concurrency. This chapter surveys some of the approaches that have been suggested and attempts to organize them coherently. The survey is by no means exhaustive; rather, the presentation emphasizes certain formalisms that exemplify the characteristics associated with other, similar formalisms.

2.1. Introduction

Approaches to formalizing the description and verification of concurrent programs fall roughly into three categories, based on the level of abstraction of the objects being described. In *logical* formalisms the objects of analysis are programs; examples of logical formalisms include various programming logics, including assertion-based logics and temporal logic. *Linguistic* formalisms, on the other hand, describe operators that can be used to build concurrent programs; examples include Hoare's

Communicating Sequential Processes, or CSP, and Milner's Calculus for Communicating Systems, or CCS. *Semantic* formalisms describe the mathematical foundations for concurrency and include as examples powerdomains, traces and synchronization trees.

The rest of the chapter is organized along the following lines. Section 2.2 describes logical formalisms in more detail and temporal logic in particular. Section 2.3 examines linguistic formalisms, emphasizing CSP and CCS. The final section presents an overview of various semantic formalisms, including powerdomains.

2.2. Logical Paradigms for Concurrency

This section reviews several of the logic-based methodologies for verifying properties of concurrent programs. The methodologies are divided into two groups, *assertional* and *modal*; each group is described in general terms, and then specific examples are given. The account here is by no means complete; it is intended rather to give a flavor of these types of formalisms and to give the reader some familiarity with terms that appear later in the thesis.

2.2.1. Assertional Logics

Assertional logics allow the verification of programs to be conducted by encoding information about a program's state at a particular point in its execution as assertions in first-order logic involving variables used in the program. These logics have been used extensively to reason about sequential programs [Gries 81], and much work has been done generalizing them to handle concurrency [Hooman and deRoever 86]. In the case of sequential programs these logics represent the operational semantics of statements in the programming language as axioms describing the action of statements on states; $\{P\}\ S\ \{Q\}$ means that in a state satisfying proposition P, statement S will, upon termination, produce a state satisfying proposition Q. To prove that a program T satisfies specification Q from initial state P, then, one proves a theorem $\{P\}\ T\ \{Q\}$ using the axioms and the rules of inference associated with the logic. The assertion P is called the *precondition* of the program, while Q is called the *postcondition.*

In the case of concurrent programs, a natural generalization of sequential assertional reasoning involves proving such a theorem for each process in isolation; however, one must ensure in doing so that the action of one process does not invalidate the proof of another process. This aspect of concurrent program verification, called *interference*, complicates the process of proving properties about concurrent programs and has spawned a variety of means for coping with it. Several researchers have handled interference by making its absence an explicit proof obligation. In their proof system for a concurrent programming language with shared variables, Owicki and Gries [Owicki and Gries 76] require that processes be verified in isolation and that each statement in each process then be shown not to invalidate the assertions in proofs of other processes. Levin and Gries [Levin and Gries 81] adopt similar proof requirements in their system for verifying CSP programs, they also add an additional proof obligation to establish that communications satisfy the assumptions made about them in the proofs of processes. Lamport and Schneider [Lamport and Schneider 84] obviate the interference problem by requiring that the only assertions allowed be invariance relations (i.e. relations whose validity are not affected by the execution of any process). However, establishing that a property is invariant requires proving the invariance for each process that might subsequently be added to a network. The system of Apt, Francez and deRoever [Apt, Francez and deRoever 80] for verifying CSP programs handles interference by restricting the variables that may appear in the verification of individual processes to the local variables of the process. Once processes are verified in isolation, a proof obligation called the *co-operation test* establishes that the values passed between processes satisfy the requirements placed on them by the sending and receiving processes.

Others have approached the issue of interference by developing *compositional* proof systems, where a compositional proof system allows one to deduce the properties of a network from the properties of the components of the network. Soundararajan and Dahl [Soundararajan and Dahl 82] present a logic for verifying CSP programs that associates with each process a history variable representing the sequence of communcations performed by the process during the course of its execution. In the proof system a predicate, *compat*, indicates whether histories are compatible (i.e. whether values received by a process have been sent by another, and vice versa). Processes are verified independently using assertions that refer only to local variables; to compose the specifications of processes, one determines if the histories are compatible and if so takes the conjunctions of specifications of the components. Misra and Chandy [Misra and Chandy 81] take a different approach in their verification framework for CSP. Their methodology is invariant-based; a process is specified in terms of two assertions, where the first assertion is gurararanteed to hold after each communication provided that the second assertion held of each previous communcation. The composition rule allows the specification of a network of processes as the conjunctions of the corresponding assertions in the specifications of each component process. Brookes [Brookes 86] adopts yet another approach by defining the composition of specifications of processes as a distinct operator on assertions. A specification of a program roughly consists of the program's statements with two assertions appearing between each statement, with one assertion at the beginning and one assertion at the end of the program. To compose processes one treats a statement and the assertions immediately surrounding it as a unit and calculates the tree of interleavings of units. This "tree" is the specification of the network of processes. In order to determine if the postcondition of the network holds after the network executes, one must determine that the pairs of assertions between statements are compatible. Brookes' system, unlike the others mentioned, is complete. Widom [Widom, Gries and Schneider 86] identifies a simple set of axioms necessary and sufficient to render so-called "trace-based" logics, like the Misra and Chandy system, complete.

2.2.2. Modal Logics

Modal logics, and in particular temporal logic, have been studied extensively as tools for expressing and verifying properties of concurrent programs. These logics enrich the propositional and predicate calculi with formula constructors, called *modal operators*, that enable the expression of properties involving the notions of possibility and necessity. Several axiomatizations have been proposed for general modal logics, each axiomatization capturing different intuitions about the behavior of possibility. Notable examples of such axiomatizations include the S4 and S5 schemes [Hughes and Cresswell 68].

The next section describes temporal logic in more detail. The syntax of formulas is presented, the semantics described and an axiomatization developed. Ways in which temporal logic has been used to specify and verify properties of programs will then be briefly described. The section following mentions other modal logics that have been suggested as programming logics.

2.2.2.1. Temporal Logic

Temporal logic is a modal logic that enables one to make assertions about the possibility and necessity of truth of propositions in the past and in the future. The usage of these logics in computer science generally involves only the future operators [Pnueli 77], so while work has been done on using past operators in programming this section will concentrate on future operators.

2.2.2.1.1. Syntax and Semantics

Different presentations of temporal logic feature different modal operators; what follows is a collection of some of the most common operators that have been used. It is necessary to postulate first a set Φ of "atomic propositions" that represent the smallest units of reasoning. Given $\phi \in \Phi$ the following grammar describes the structure of formulas we shall consider.

$$
\begin{array}{lll}
\Gamma & ::= & \phi \\
 & | & \neg\Gamma \\
 & | & \Gamma \vee \Gamma \\
 & | & \Box\Gamma \\
 & | & \mathrm{O}\,\Gamma
\end{array}
$$

Intuitively the propositions have the following meaning. An atomic proposition denotes some assertion about the possible state of a program. The propositional operators have the usual meaning. $\Box\Gamma$ states that Γ holds throughout the execution of the program. $\mathrm{O}\Gamma$ means that Γ becomes true at the next step in the execution of the program.

Other notations, including other proposition connectives such as $\wedge$ and $\Rightarrow$ and modal operators such as $<>$, can be defined in terms existing operators. In particular, $<>\Gamma$, which is the dual of $\Box\Gamma$ and can be written as $\neg\Box\neg\Gamma$, means that Γ becomes true at some point in the execution of the program.

The formal semantics of temporal propositions can be given on the basis of sequences of states, which sequences intuitively correspond to execution histories of programs. To this end, let S be the set of states, and let $s_1, s_2, \ldots \in S$. Let $\sigma = s_1 s_2 s_3 \ldots$ be a sequence of states, and let σ_i denote the suffix of σ headed by the $i^{\underline{th}}$ element in σ, so that $\sigma_1 = \sigma$ and $\sigma_i = s_i s_{i+1} \ldots$. The elements of Φ will be interpreted as functions of type $S \rightarrow \{True, False\}$. The following constitutes a semantic interpretation of the previous formulas.

$\sigma \models \phi$ if and only if $\phi(s_1) = True$.

$\sigma \models \neg\Gamma$ if and only if it is not the case that $\sigma \models \Gamma$.

$\sigma \models \Gamma_1 \vee \Gamma_2$ if and only if $\sigma \models \Gamma_1$ or $\sigma \models \Gamma_2$.

$\sigma \models \Box\Gamma$ if and only if for all $i > 0$, $\sigma_i \models \Gamma$.

$\sigma \models \mathrm{O}\Gamma$ if and only if $\sigma_2 \models \Gamma$.

In general a concurrent program will be identified with its set of state sequences, and a temporal formula will hold of a program if it holds of each of its traces.

2.2.2.1.2. An Axiomatization of the Logic

The following list of axioms and inference rules due to Pnueli [Pnueli 79] includes the S4 axioms for $\Box$ and axioms for O. They also indicate that $\Box$ is the "fixpoint" of O.

A1. $\Box(P \Rightarrow Q) \Rightarrow \Box P \Rightarrow \Box Q$

A2. $\Box P \Rightarrow P$

A3. $O(\neg P) \Leftrightarrow \neg O(P)$

A4. $O(P \Rightarrow Q) \Rightarrow OP \Rightarrow OQ$

A5. $\Box P \Rightarrow OP$

A6. $\Box P \Rightarrow O\Box P$

A7. $\Box(P \Rightarrow OP) \Rightarrow (P \Rightarrow \Box P)$

R1. If P is an instance of a classical tautology then $\vdash P$.

R2. If $\vdash P$ and $\vdash P \Rightarrow Q$ then $\vdash Q$.

R3. If $\vdash P$ then $\vdash \Box P$.

2.2.2.1.3. Using Temporal Logic as a Verification Tool

A wealth of methodologies based on temporal logic as a verification tool have been proposed. It is beyond the scope of this thesis to address these in detail; the interested reader is referred to the survey paper of Pnueli [Pnueli 86]. Some general remarks, however, can be made. These methodologies usually involve defining a model of concurrent programs and a *satisfaction* relation between objects in the model and temporal formulas is defined and axiomatized. To prove that an object in the model has a particular temporal property, one typically proves that the object satisfies the property.

Much work has also been done in identifying interesting varieties of temporal properties. In particular, the classifications of *safety* and *liveness* properties [Lamport 77] have inspired much research. Intuitively, a safety property dictates that "nothing bad happen," while a liveness property requires that "something good happen" [Owicki and Lamport 82].

2.2.2.2. Other Modal Logics

Other modal logics have been examined in the context of reasoning about concurrent programs. *Concurrent dynamic logic* [Peleg 84] allows the possibility operator to be subscripted by a program expression; $<p>P$ states that if program p terminates then it may be in a state satisfying proposition P. Program expressions have a syntax that includes a parallelism construct.

Logics of *knowledge* have been used to reason about distributed systems [Halpern and Moses 84, Chandy and Misra 85]; only recently, however, have they been considered as a tool for verifying programs [Beck and Cleaveland 87]. A logic of knowledge includes a knowledge operator, K, in its syntax of propositions; KP holds of a process at a certain point in its execution if P is true of any possible system execution that includes the history of the process in question up to that point.

2.3. Linguistic Models of Parallelism

This section describes two abstract concurrent programming schema, Hoare's Communicating Sequential Processes and Milner's Calculus for Communicating Systems. Numerous other concurrent programming languages exist; we choose to elaborate on these, however, since they abstract away most details not relating to concurrency.

2.3.1. Communicating Sequential Processes

The Communicating Sequential Processes formalism due to Hoare [Hoare 78, Hoare 85] describes an abstract language for representing processes and a logical framework for reasoning about their correctness. Processes are defined in terms of the events they may engage in and the courses of action they may pursue after engaing in events. The definition of certain process constructors also allows new processes to be constructed from existing ones; the behaviors of these constructors have equational descriptions that enable the definition of a relatively simple verification scheme [Hoare 85]. The remainder of this section describes the CSP language of [Brookes, Hoare and Roscoe 84] in more detail.

In the following discussion, $A, B, \ldots$ will refer to the sets of events a process may perform, and $a, b, \ldots$ will refer to elements of $A, B, \ldots$. The grammar in figure 2.1 defines the syntax of CSP expressions. Operationally, these process constructors have the following informal descriptions. *STOP* engages in no events; it represents a terminated process. The process $a \rightarrow P$ first engages in event a and then behaves like process P.

The next three process constructors presented in figure 2.1 define different notions of choice. The process $P|Q$ may initially perform any of the events initially available to either P or Q, assuming the initial events of P and Q are disjoint; depending on the event chosen, $P|Q$ then behaves like either P or Q. The process $P \sqcap Q$ behaves like either P or Q; the choice is

P ::=	$STOP$	
|	$a \rightarrow P$	
|	$(P \mid P)$	
|	$P \cap P$	
|	$P+P$	
|	$P \backslash C$	$C \subseteq A$
|	P/s	$s \in A^*$
|	$f(P)$	$f \in A \rightarrow B$
|	$\mu x{:}A.F(x)$	if $F(x)$ is an expression containing x free
|	$P \| P$	
|	$P \||| P$	

Figure 2.1. The Process Constructors of CSP.

made nondeterministically. $P+Q$ behaves like $P|Q$ except that the initial events available to P and Q need not be disjoint; if a is available to both P and Q then a is available to $P+Q$, and the continuation after a occurs is either the continuation available to P or the continuation available to Q, with the choice occurring nondeterminstically.

If $C \subseteq A$ then $P \backslash C$ represents the hiding of events in C occurring in P. That is, $P \backslash C$ behaves like P except that events in C appear not to occur. If s is a string of events then P/s denotes the process that behaves like P would after after engaging in s. The process $f(P)$ is the process P with the events renamed according to f. The process $\mu x{:}A.F(x)$ represents a recursively defined process with event set A that satisfies the equation $x = F(x)$. Certain recursive definitions that do not denote unique processes are possible; an example is $\mu x{:}A.x$, which may be any process at all, since any process satisfies $x = x$. Such a process is called a *divergence* in CSP terminology.

The final two constructors represent synchronous and asynchronous parallel execution. The process $P \| Q$ performs in lock-step the parts of P and Q that perform the same events at the same time; $P|||Q$ performs interleavings of the events of P and Q.

The formal specification of CSP process operators takes the form of equationally based laws that the operators must satisfy. The complete set of these laws will not be duplicated here, but the following should convey their flavor.

$$P + Q = Q + P$$

$$Y = F(Y) \Leftrightarrow Y = \mu X{:}A.F(X)$$

$$P \| STOP = STOP$$

CSP also has a denotational semantics given in terms of the sequences of events that a process may perform, the actions a process may refuse after performing a sequence, and the possibility of a process becoming divergent after executing a sequence. This semantic specification, called the *failures* model for CSP, will be elaborated on later in the thesis.

2.3.2. The Calculus for Communicating Systems

The Calculus for Communicating Systems (CCS) is an algebraic formalism for reasoning about concurrent and nondeterministic processes. A complete presentation of the calculus may be found in [Milner 80]; enrichments of the formalism (the extended formalism is called SCCS, for Synchronous Calculus of Communicating Systems) are described in [Milner 83]. Like CSP, CCS enables the description of processes in terms of the actions they may perform, including synchronization and communication with other processes. Action sets are required to have a distinguished action τ and an *inverse* action for each non-τ action; the intent is that actions may synchronize with their inverses, wtih τ being the result of the synchronization. In general, processes may choose from several different actions at any point during their computations, and processes may also exhibit nondeterministic behavior. The number of different choices available to a process at any given point in the process execution is finite.

A CCS action set A must satisfy the following two conditions.

(i) There is a distinguished action, τ, in A.

(ii) If $a \in A$ and $a \neq \tau$ then so is $\underline{a}$, and $\underline{\underline{a}} = a$.

τ is called the "silent" action. Intuitively, one may think of τ as being an internal transition within a process or a communication between two different component processes of a system. The actions a and $\underline{a}$ are inverses of each other and represent actions which can synchronize.

Given A, the CCS terms are described by the grammar in figure 2.2. The

$$
\begin{array}{rlll}
P ::= & NIL & & \\
| & aP & a \in A & \\
| & P+P & & \\
| & P|P & & \\
| & P \backslash S & S \subseteq A & \\
| & P[L] & & \\
| & fix(x.b) & x \text{ free in } b &
\end{array}
$$

Figure 2.2. The Process Constructors of CCS.

NIL process performs no actions and terminates. aP is a process which performs an a action and then behaves like P. $P+Q$ may behave like P or Q. Initially, the actions available to $P+Q$ include all the actions initially available to either P or Q, and the courses of action that $P+Q$ may pursue after an initial action are exactly the courses of action available to P or Q after the same initial action.

$P|Q$ represents the concurrent execution of P and Q and corresponds to the process whose courses of action roughly represent the interleaved courses of action available to P or Q together with appropriate synchronizations. Initially, this composite process may perform any first action available to P or Q, after which it may perform either the first action of the process not selected in the first step or the second action available in the process which was selected in the first step. Additionally, at any step in the computation, if P may perform an action a and Q may perform $\underline{a}$, the inverse of a, then $P|Q$ may perform τ, a *synchronization*; the next actions available to $P|Q$ after a τ are the next actions available to P and Q. $P\backslash S$ corresponds to the restriction of P with respect to a set S of actions; this process behaves like P except that it may not perform any sequence of actions headed by an action in S or an action whose inverse is in S. If A_1 and A_2 are two action sets and $L{:}A_1 \rightarrow A_2$ is a mapping such that $L(\tau) = \tau$ and $L(\underline{a}) = \underline{L(a)}$ for any $a \in A_1$, then L is called a *relabeling* and $P[L]$ is the process resulting from the relabeling of the actions in P by L. The process $fix(x.b)$ defines a process recursively. The subterm b is required to be *guarded*, which is to say that there must be an action preceding any occurence of x in b. This requirement precludes the existence of the divergences in CCS.

We shall have occasion to refer to the SCCS composition operator, $\otimes$, as defined in [Milner 83]. $\otimes$ is a synchronous parallel composition; in $P \otimes Q$ the actions in $P \otimes Q$ are required to happen in lock-step synchrony. To define it we first strengthen the conditions saitsfied by the A in the following fashion. We require that $(A, 1, \_, \circ)$ be an Abelian group, with $a \circ \underline{a} = 1$ representing the synchronization of a and $\underline{a}$ and $a \circ b$ representing the lock-step execution of a and b.

The formal semantics of CCS are given in terms of *transition relations* that specify when a process may perform an action. Figure 2.3 describes these relations as in [Milner 80]. In general $P \rightarrow^a P'$ means that process P performs action a and "evolves" into process P'.

Given the formal semantic definition of CCS, an important property of the concurrent composition operator is captured in the following theorem

$NIL \rightarrow^{a} P$	for no P.
$aP \rightarrow^{a} P$	
$P \rightarrow^{a} P'$	$\Rightarrow P+Q \rightarrow^{a} P'$
$Q \rightarrow^{a} Q'$	$\Rightarrow P+Q \rightarrow^{a} Q'$
$P \rightarrow^{a} P'$	$\Rightarrow P \vert Q \rightarrow^{a} P' \vert Q$
$Q \rightarrow^{a} Q'$	$\Rightarrow P \vert Q \rightarrow^{a} P \vert Q'$
$P \rightarrow^{a} P',\ Q \rightarrow^{\underline{a}} Q'$	$\Rightarrow P \vert Q \rightarrow^{\tau} P' \vert Q$
$P \rightarrow^{a} P',\ a \notin S,\ \underline{a} \notin S$	$\Rightarrow P \backslash S \rightarrow^{a} P' \backslash S$
$P \rightarrow^{a} P'$	$\Rightarrow P[L] \rightarrow^{L(a)} P'[L]$
$b[fix(x.b)/x] \rightarrow^{a} P'$	$\Rightarrow fix(x.b) \rightarrow^{a} P'$
$P \rightarrow^{a} P',\ Q \rightarrow^{b} Q'$	$\Rightarrow P \otimes Q \rightarrow^{a \circ b} P' \otimes Q$

Figure 2.3. Operational Semantics for CCS.
$a, b \in A$; P, Q, P', Q' are processes.

due to Milner [Milner 80]. $\Sigma a_i P_i$ is shorthand for $a_1 P_1 + \ldots + a_n P_n$, and $\Sigma \tau (P_i | Q_j)$(if $a_i = \underline{b_j}$) represents the case of $a_i P_i$ and $b_j Q_j$ when $a_i = \underline{b_j}$.

Expansion Theorem:

If $P = \Sigma a_i P_i$ and $Q = \Sigma b_j Q_j$ then

$P|Q = \Sigma a_i(P_i|Q) + \Sigma b_j(P|Q_j) + \Sigma \tau(P_i|Q_j)$(if $a_i = \underline{b_j}$).

Thus concurrent execution may be correctly interpreted in the CCS formalism by nondeterministic interleaving of the atomic actions and allowable synchronizations of the processes executing in parallel.

The CCS formalism supports an algebraic style of reasoning. Equivalences between processes are defined and axiomatized, and reasoning is carried out equationally. Milner [Milner 80] describes two notions of process equivalence. The first is *strong* equivalence, and it is defined as the limit of the following hierarchy of nested relations, where P and Q are arbitrary processes.

$$P \sim_0 Q.$$

$$P \sim_{n+1} Q \Leftrightarrow \forall a \in A.$$

$$P \rightarrow^a P' \Rightarrow \exists Q'.\, Q \rightarrow^a Q' \;\&\; P' \sim_n Q' \;\&$$

$$Q \rightarrow^a Q' \Rightarrow \exists P'.\, P \rightarrow^a P' \;\&\; P' \sim_n Q'.$$

P and Q are *strongly equivalent*, or $P \sim Q$, if and only if, for all n, $P \sim_n Q$.

The second equivalence, *observational* equivalence, is essentially an extensional version of strong equivalence. Defining it requires introducing the notions of *experiment* and *observable experiment*. Define the set of experiments to be A^*; we then say that $P \rightarrow^s P'$ if $s = \varepsilon$ and P and P' are identical or if $s = as'$ for $a \in A$ and $s' \in A^*$ and there is a P'' such that $P \rightarrow^a P''$ and $P'' \rightarrow^{s'} P'$. Define the set of observable experiments as $(A - \{\tau\})^*$, and let $s \in (A - \{\tau\})^*$ be defined by $s = s_1 \ldots s_n$, where each $s_i \in A - \{\tau\}$. Process P admits an observable experiment s and transforms to P' (written $P \Rightarrow^s P'$) if there is an $s' \in A^*$ and integers $m_0, \ldots, m_n$ such that $s' = \tau^{m_0} s_1 \tau^{m_1} \ldots \tau^{m_{n-1}} s_n \tau^{m_n}$ and $P \rightarrow^{s'} P'$. We may now define a nested hierarchy of relations between processes whose infinite intersection will be observational equivalence. Let P and Q be arbitrary processes.

$$P \approx_0 Q.$$

$$P \approx_{n+1} Q \Leftrightarrow \forall s \in (A - \{\tau\})^*. \quad P \Rightarrow^s P' \text{ implies } \exists Q'.\, Q \Rightarrow^s Q' \;\&\; P' \approx_n Q' \;\& \quad Q \Rightarrow^s Q' \text{ implies } \exists P'.\, P \Rightarrow^s P' \;\&\; P' \approx_n Q'.$$

P and Q are *observationally equivalent*, or $P \approx Q$, if and only if, for all n, $P \approx_n Q$. Both strong and observational equivalence are axiomatized equationally in [Milner 80].

Congruences also play an important role in the proof theory. Briefly, an equivalence $=$ is a congruence if, when $C[]$ is a context and $p = q$, then $C[p] = C[q]$. Strong equivalence is a congruence, but observational equivalence is not. Accordingly, in [Milner 80] observational congruence, which is the weakest congruence that is strictly finer than observational equivalence, is defined and axiomatized equationally.

One model for CCS describes processes as *synchronization trees*, which are unordered, potentially infinite, finitely branching trees with edges labeled by actions. Figure 2.4 gives examples of CCS expressions represented as synchronization trees. At each node of the tree the branching represents the possible choices to the process being modeled, while the paths in the tree represent possible execution sequences available to the process represented by the synchronization tree. Each CCS operation on processes has a corresponding tree-theoretic operation. These trees are formalized categorically by Winskel [Winskel 83a].

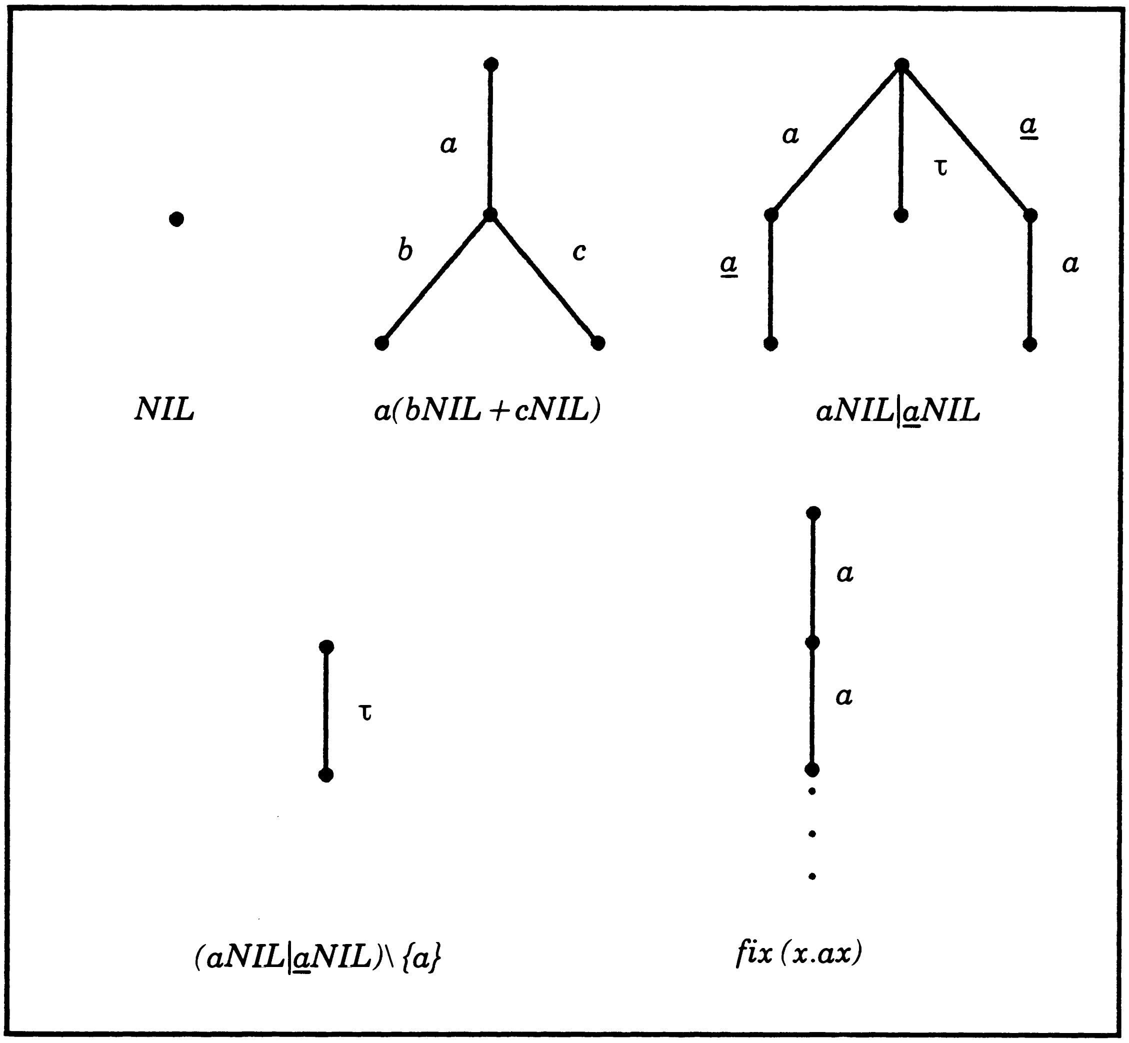


Figure 2.4. Examples of Synchronization Trees.

2.4. Semantic Models of Concurrency

The next few sections describe mathematical structures that have been used to give meaning to concurrent computation. The structures are denotational in the sense that concurrent agents are given denotations in them.

2.4.1. Trace-Based Models of Concurrency

Traces have been used extensively to describe the semantics of modal logics and programming languages. Informally, a trace represents a partial execution history of a process; it usually consists of a sequence of states or events that processes encounter during their execution. A process is then identified with the set of its traces. The next sections describe a general trace model and then examine alterations to the model that have been suggested.

2.4.1.1. Traces

The formalization of traces we are about to present is event-based in the sense that the building blocks of traces will be events, or actions, in a process's history. Other trace models are state-based [deNicola and Hennessy 82, Brookes, Hoare and Roscoe 84] and have different consistency restraints than the one presented here; however, what follows should convey the essence of trace models.

Let A be the set of events a process may engage in. For example, A may be a set of atomic actions processes may execute. A *trace* is a (potentially empty) finite sequence of elements of A. In general we shall write trace t as $a_1a_2...a_n$, where each $a_i \in A$. The set of all traces will be denoted A^*.

A process can now be represented as a set of traces that contains the empty trace and is prefix-closed. These two conditions reflect our interpretation of a trace as a partial execution of a process. Every process must have a starting point, a point at which no computation has been performed, and this assumption leads to the inclusion of the empty trace in each process representation. Also, if a process has executed a string of actions then it has executed every initial string of actions contained in the string; this fact accounts for the prefix closure property.

Traces form a complete partial order under the prefix ordering, where for traces s and t, $s \leq_p t$ if and only if s is a prefix of t. Process representations also form a complete partial order under the set inclusion ordering; given process representations P and Q, $P \leq_I Q$ exactly when $P \subseteq Q$.

2.4.1.2. Failures Models

An enrichment of the basic traces models is the *failures* model used to give semantics to CSP [Brookes, Hoare and Roscoe 84, Brookes and Roscoe]. Intuitively, processes are represented as sets of failures, where a failure is a pair consisting of a trace and a *refusal*, which is a finite set of events that a process may refuse to perform after executing the associated trace.

Formally, let A be the set of events. For $s \in A^*$ let the relation $P \rightarrow_s P'$ denote that P may perform trace s and thereafter behave like Q, and let

$$initials(P) = \{a \in A \mid \exists P'.\ P \rightarrow_a P'\}.$$

Also let ε denote the empty trace. Then

$$refusals(P) = \{X \subseteq A \mid X\ finite\ \&\ \exists P'. P \rightarrow_\varepsilon P'\ \&\ X \cap initials(P') = \varnothing\}.$$

That is, *refusals(P)* represents the sets of events a process P may refuse initially. Process P will be now identified with its set of *failures*, where a failure is a pair whose first component is a trace and whose second component is a refusal.

$$failures(P) = \{<s, X> \mid \exists P'.P \rightarrow^{s} P' \; \& \; X \in refusals(P')\}$$

Equality between processes is defined as equality between their failure sets. This representation also induces a complete partial ordering on processes via set inclusion.

An improvement of the model that handles divergences is given by Brookes and Roscoe [Brookes and Roscoe].

2.4.1.3. A Testing Model

The model described here derives from the work of de Nicola and Hennessy [deNicola and Hennessy 82]. Processes are defined in terms of their traces, the set of events that a process *may* perform after executing a trace, and the set of states a process *must* perform after executing a trace. As before, let the set A denote the set of events, and let $P \rightarrow^{s} P'$ when P may perform trace s and thereafter behave like P'. Also define *initials(P)* as before. Now let

$$May(P, s) = \{a \in A \mid \exists P'.P \rightarrow^{s} P' \; \& \; a \in initials(P')\}$$

and

$$Must(P, s) = \{a \in A \mid \forall P'.P \rightarrow^{s} P' \Rightarrow a \in initials(P')\}.$$

A process is identified with its traces and its must and may sets in the following fashion.

$$Test(P) = \{<s, M, N> \mid \exists P'. P \rightarrow^{s} P' \; \& \; M = Must(P, s) \; \& \; N = May(P, s)\}.$$

The model presented in [deNicola and Hennessy 82] is more complex and enables the handling of diverges; it also has a complete partial ordering associated with it.

2.4.1.4. An Acceptance Model

Hennessy [Hennessy 83] describes a model of concurrency based on trace-finite set pairs. As the model is very similar to a model presented in chapter 4 of this thesis, we delay its description until then.

2.4.2. Tree-Based Models

Synchronization trees have been proposed as a means of specifying the semantics of CCS [Milner 80] and more generally as a category-theoretic model of concurrency [Winskel 83a]. Intuitively, synchronization trees represent a process as a tree whose arcs are labeled by the events a process may perform at the stage in its computation corresponding to the level in the tree. As they were discussed briefly earlier, we shall not say more about them here.

2.4.3. Non-Well-Founded Set-theoretic Models

Another model for concurrent computation uses non-well-founded sets to model processes. This style of semantic model, developed in this thesis and by Aczel [Aczel 85], is event-based and models processes as sets of pairs, where the first element of each pair is an event the process may engage in and the second element represents the remainder of the computation the process follows after performing the associated event. These models are described in greater detail in chapter 5 of this thesis.

2.4.4. Domain Theory

This section discusses several domain-theoretic approaches to modeling parallelism. Each of these approaches specifies a *powerdomain*, which is a mathematical structure used to model nondeterministic computation, and hence each depends on the correspondence between nondeterminism and concurrency. Implicit in each of these powerdomain constructions is the belief that the denotation of a program should reflect all possible results of the program; therefore, each of these powerdomains reflects some families of sets back into the powerdomain. Cardinality, of course, prohibits the reflection of all sets of a particular domain within itself, and much of the work involved in construction these powerdomains centers around defining an appropriate collection of sets to be reflected. In what follows D will refer to a countable domain with $\leq_D$ being the ordering on D, and S represents the powerset of D.

2.4.4.1. The Egli-Milner Ordering and Plotkin Powerdomain

Defining a powerdomain in the same style used to define domains for sequential computation requires the notions of successive approximation and limit, and these in turn require the concept of an ordering on sets of elements in a domain. The Egli-Milner ordering represents one well-studied ordering, and it forms the basis for the Plotkin powerdomain constructions [Plotkin 76]. This ordering meets two natural requirements that one wishes to place on set approximations, namely, that elements in a set may "improve" in more than one way and that new elements may be introduced into the set as the result of an improvement of an element that already exists in the set. More formally, given two sets X and Y that are subsets of D, $X \leq_{EM} Y$ if and only if $\forall x \in X . \exists y \in Y . x \leq_D y$ & $\forall y \in Y . \exists x \in X . x \leq_D y$, so that $X \leq_{EM} Y$ if and only if each element of X is improved by an element in Y and each element in Y improves an element in X.

The intuitive appeal of this ordering makes it appear reasonable as an ordering on sets of elements in D; care must be taken, however, in defining the class of sets to be included in the powerdomain construction because the ordering is not a partial order on S, due to the fact that it is not anti-symmetric over all subsets of D. This problem can be circumvented, however, by defining an equivalence relation $\sim$ where for $X, Y \in S$, $X \sim Y$ exactly when $X \leq_{EM} Y$ and $Y \leq_{EM} X$. The structure $(S/\sim, \leq_{EM})$ is now a partial order.

This structure turns out to correspond to Plotkin's powerdomain constructor, P. On a flat domain D, $P(D)$ includes sets that are either finite or contain $\perp$, reflecting the assumption that processes are finitely branching and hence any infinite set of results has to include (because of Konig's lemma) the nonterminating computation. The appropriate sets may also be characterized as the *finitely generable* subsets of a domain D

(which are the images of continuous functions from $\{0,1\}^* + \{0,1\}^\infty$ into D over $\{0,1\}^\infty$) and thus useful in solving recursive domain equations. Another interesting property of these sets is that they are convex-closed, meaning that if x and y belong to X which is in $P(D)$ and there is an a in D such that $x \leq_D a$ and $a \leq_D y$ then a also belongs to X. In general P is valid for domains satisfying the SFP property.

2.4.4.2. The Smyth Powerdomain

The Smyth powerdomain is intended as a semantic basis for proof systems for total correctness of concurrent systems. The ordering in the powerdomain is weaker than the Egli-Milner ordering; if X and Y are subsets of D then $X \leq_S Y$ exactly when $\forall y \in Y. \exists x \in X. x \leq_D y$. The sets in this powerdomain are the finitely generable sets quotiented by the equivalence $=_S$, where $X =_S Y$ if and only if $X \leq_S Y$ and $Y \leq_S X$. It is easy to see that any set containing $\perp$ is equivalent to $\{\perp\}$, meaning that processes which may diverge cannot be discerned from processes that must diverge. Thus, the only objects that exhibit interesting behavior are the ones which must terminate.

2.4.4.3. The Hoare Powerdomain

The Hoare powerdomain construction [Hennessy 82] is intended to provide the semantics for Hoare-style proofs systems for reasoning about partial correctness and is dual to the Smythe powerdomain. $H(D)$ includes the downward closed subsets of D; hence, $X \in H(D)$ exactly when for each element $x \in X$, if $y \leq_D x$ then $y \in X$. Each nonempty set of $H(D)$ must contain $\perp$, note. Also, if x is in D and $x\downarrow$ is defined as $\{y \in D | y \leq_D x\}$ then $x\downarrow$ is in $H(D)$ and if $y \leq_D x$ then $y\downarrow \leq_H x\downarrow$. Its should be noted that the sets included by this powerdomain construction are a proper subset of the sets defined by the Plotkin powerdomain construction.

The sets included by $H(D)$ may be characterized by the class of continuous predicates mapping D to the two element domain $\{\perp, \top\}$ in that for such a predicate f, $f^{-1}(\perp)$ is downward closed and completely specifies the function.

Chapter 3

The Nuprl System

The major aim of this thesis involves the development of a framework for reasoning formally in an automated fashion about concurrency, and we propose that using an automated proof assistant such as the Nuprl is one natural way to do so. Nuprl is a system for developing formal constructive mathematics that embodies a proof editor, a scheme for defining notation and a formalized metalanguage. It is in the context of this system that the models of concurrency presented later are developed, so this chapter presents a broad overview of the features of the Nuprl mathematical language, constructive type theory, and the Nuprl system.

3.1. Introduction

The Nuprl system has borrowed from work done on other systems and theories that formalize mathematics. Thus, before describing the general features that make Nuprl a suitable system to use for our purposes we shall mention some of the other work that has also been done in these areas.

The logical language of the system, constructive type theory, embodies a constructive foundational theory of mathematics that is intuitionistic in the sense of Brouwer [Brouwer 23] and is based on the type theories developed by Martin-Lof [Martin-Lof 82]. The logic induced by this type theory is constructive in the sense that the law of the excluded middle does not always hold; other areas of mathematics, including arithmetic [Heyting 30] and real analysis [Bishop 67], have been formalized in such a constructive style. The constructiveness of the Nuprl logic enables proofs to be interpreted as programs [Bates and Constable 85], and programming can therefore be seen as a specialized branch of proof theory. Other research being conducted in machine-assisted formal theories include the Automath project [deBruijn 80], the LCF project [Gordon, Milner and Wadsworth 79], the Calculus of Constructions project [Coquand and Huet 85] and the Logical Framework project [Harper, Honsell and Plotkin 87].

The Nuprl system interface makes Nuprl a feasible forum for developing formal mathematics. This interface supplies the user with a proof editor, a facility for extending notation and an interface between the proof editor and the ML programming language that allows more complex rules of inference to be constructed from the primitive inference rules supplied by the system. The ideas behind so-called the "refinement-style" proof editing editing feature in Nuprl were influenced heavily by the work of the Cornell Program Synthesizer group [Teitelbaum and Reps 81], while the notion of having a programming language as a metalanguage in which to write more complex proof rules based on the primitive ones was borrowed in large part from the LCF project.

Understanding Nuprl involves understanding its type theory and understanding the system interface. Accordingly, the rest of the chapter is organized in the following fashion. Section 3.2 describes the general features of type theory, while section 3.3 focuses on the Nuprl type theory in more detail. The final section details the features of the Nuprl system.

3.2. Type Theory

Set theory and type theory are both formal theories for reasoning about collections of objects. Sets and types, however, have fundamentally different interpretations, and the differences between these interpretations yield fundamentally different theories. A set is a collection of objects whose only common feature is that they belong to the same set; furthermore, equality between objects in a set and between sets is defined universally via the extensionality axiom of Zermelo-Fraenkel set theory. A type, on the other hand, is a collection of objects exhibiting a *common structure* and an *equality relation* describing the conditions under which objects in a type may be said to be equal. The nature of the structure that objects of a certain type have and of the equality relation between elements in a type devolves from the structure of the type, for type theories usually embody an inductive definition of what a type is. For example, if A and B are types then it is typically the case that the *Cartesian product* of A and B, denoted $A \times B$, is also a type whose members are pairs whose first component comes from type A and the second component comes from type B; its equality relation generally states that two pairs are equal exactly when their corresponding components are equal.

The Cartesian product is an instance of a *type constructor*, an operator that allows more complex types to be constructed from simpler ones. Another example of a type constructor is the disjoint union constructor, which is usually written as $A|B$ for types A and B. The elements of $A|B$ usually consist of pairs comprising an element of either A or B and a tag denoting which type, left or right, the first component comes from (thus $A|A$ and A are distinct types); two elements in $A|B$ are equal if they have the same tags and their other components are equal in the appropriate type. Similarly, the function space of A into B, denoted $A \rightarrow B$ for types A and B, is a type whose elements are functions from A to B and whose equality may, for example, be extensional functional equality. In addition to these three type constructors, most type theories also have more complex type constructors that we shall describe later.

3.2.1. Specifying Type Theories

Type theories are usually specified on the basis of a collection of terms and a computation system given by a set of rewriting rules on these terms. Intuitively, some of these terms will denote type expressions in the theory, while other terms will correspond to elements which are members of types. For instance, in a type theory having a type *int* of integers and a function space constructor $\rightarrow$, $int \rightarrow int$ is a term which represents the type of integer-valued functions on the integers, while the term $\lambda x.x+1$ represents an element in *int→int* that for any integer computes its successor. Terms also may be classified as *canonical* or *noncanonical.* Canonical terms are terms which are in *normal form* according to the computation rules for the collection of terms; that is, no computation may be performed on the term. For example, in a type theory having *int* as a type with integers as terms of type *int*, *3* is a canonical term of the theory. Noncanonical terms, on the other hand, may be computed; these terms have reducible expressions (redexes) that may be replaced according to the rules specified by the computation system. In the previously mentioned theory, for instance, $3+7$ is a noncanonical term because $3+7$ may be rewritten as a canonical term, *10*.

Given such a term-rewriting system, completely specifying a type theory involves giving criteria for when a term represents a type and when it represents an element of a given type, as well as indicating when two types are equal and when two elements in a given type are equal. One usually does so by describing the canonical terms that denote types and elements of types and the conditions under which canonical types are equal and canonical elements of a given type are equal. One then ascribes to noncanonical terms the properties of the canonical term, if one exists, to which the noncanonical term reduces. Thus, in the example mentioned previously, $3+7$ has type *int* because it reduces to canonical term *10*, which has type *int*.

3.2.1.1. Canonical Types and Equality Between Types

The canonical terms which denote types usually have an inductive specification. The canonical terms representing the base types, such as the type of integers or the empty type (a special type having no elements), are given explicitly. Then, if A and B are terms representing types then terms constructed from A and B using the type constructors of the theory are canonical terms representing the appropriate type. A noncanonical term describes a type exactly when it reduces to a canonical term that denotes a type, and the type the noncanonical term represents is exactly the same as the type represented by the canonical term it reduces to. For example, in a theory having *int* as a base type, a term $\lambda x.x$, and β-reduction as a rewrite rule, the type denoted by the noncanonical term $(\lambda x.x)(int)$ is the type *int* since $(\lambda x.x)(int)$ reduces to *int*.

Equality between terms denoting types can either be *extensional* or *intensional*. Two canonical types are extensionally equal if and only if they have the same members; extensional type equality is the basis of Martin-Lof type theories [Martin-Lof 82]. On the other hand, two canonical types are intensionally equal if they have the same type constructor and the subtypes joined by the type constructor are equal types; this notion of type equality is central to Nuprl [Constable et al. 86]. In either definition two noncanonical types are equal exactly when they reduce to equal canonical types. In the previous example, the terms $(\lambda x.x)(int)$ and $(\lambda x.x)((\lambda x.x)(int))$ represent types since they both reduce to *int*, and since they both reduce to *int* they are equal as types.

3.2.1.2. Membership Criteria for Canonical Terms

Membership criteria for canonical terms in types also have an inductive flavor. The canonical terms which inhabit the base types are given explicitly; in the case of *int*, for example, the canonical terms may be the traditional decimal representations of integers. Each type constructor also has associated with it a term constructor that, when applied to appropriate terms, yields canonical terms of the constructed type. In the case of $int \to int$, the canonical member constructor associated with $\to$ might be λ-abstraction; given a term b, such as $x+1$, which may have variable x free in it, $\lambda x.b$ is a canonical term of $int \to int$ exactly when for any term t of type *int*, $b[t/x]$ (for an appropriate capture-free definition of substitution) is a term of type *int*. A noncanonical term inhabits a type if it reduces to a canonical term having that type. For example, in the previously mentioned example $(\lambda x.x+1)(4)$ inhabits *int* since it reduces to 5, which is a canonical member of *int*.

Canonical type constructors also specify the equality condition for canonical elements in the type. For example, a Cartesian product type constructor generally specifies component-wise equality as the equality relation between pairs in Cartesian product types. Noncanonical terms are equal in a type if they reduce to equal canonical elements in the type.

3.2.2. Constructions in Type Theory

Type theories typically formalize the construction and manipulation of terms representing types and elements in types, as well as the reasoning about equality of, and within, types, with a set of rules that describe the proof obligations needed to conclude the correctness of a certain conclusion. These conclusions are usually called *judgements*. Each base type and type constructor usually has a number of rules describing the conditions under which types may be constructed and manipulated and the criteria for membership in the type and equality within the type. These rules can usually be classified as *introduction* rules (*intro* rules for short), which prescribe when a type or a member of a type may be constructed or equality concluded, and *elimination* (or *elim*) rules, which give the ways in which types and members of types may be used to construct other types and members of types. For example, an intro rule corresponding to the Cartesian product type constructor might state that if A and B are types then so is $A \times B$, and an intro rule for membership in the Cartesian product type might state that if a is of type A and b is of type B then the pair $<a,b>$ is of type $A \times B$. The corresponding elim rules for the Cartesian product allow us to assume that if a type can be constructed under the assumption of the typehood of A and B then it can be constructed under the assumption of the typehood of $A \times B$ and that if a term in a type can be constructed under the assumption that term a has type A and term b has type B then it can be constructed under the assumption that the pair $<a,b>$ has type $A \times B$.

The rules associated with a type theory have a natural Post-style presentation, where the antecedents to the rule appear above the line and the conclusion appears below. We will use turnstyles ($\vdash$) to signify correctness (or provability) of associated constructions. Thus, the intro and elim rules for the Cartesian product constructor can be written as in figure 3.1 H represents a list of assumptions that may have been made in the

intro	$\dfrac{H \vdash A\ type,\ H \vdash B\ type}{H \vdash A \times B\ type}$	$\dfrac{H \vdash a\ in\ A,\ H \vdash b\ in\ B}{H \vdash <a,b>\ in\ A \times B}$
	$\dfrac{H \vdash A = A',\ H \vdash B = B'}{H \vdash A \times B = A' \times B'}$	$\dfrac{H \vdash a = a'\ in\ A,\ H \vdash b = b'\ in\ B}{H \vdash <a,b> = <a',b'>\ in\ A \times B}$
elim	$\dfrac{H,\ A\ type,\ B\ type \vdash C\ type}{H,\ A \times B\ type \vdash C\ type}$	$\dfrac{H,\ a\ in\ A,\ b\ in\ B \vdash c\ in\ C}{H,\ <a,b>\ in\ A \times B \vdash c\ in\ C}$

Figure 3.1. *Elim* Rules for the Cartesian Product Type Constructor.

course of the construction.

The rules for the function space constructor also bear examination, for the intro rules specify that assumptions may be added to the list of assumptions alluded to above. Figure 3.2 lists these rules.

3.2.3. Connecting Type Theory and Logic

The rules governing the construction of types and members of types look very similar to the natural deduction rules for constructive logic. The intro rule for establishing the correct construction of Cartesian product types, for

intro	$\dfrac{H \vdash A\ type,\ H, A\ type \vdash B\ type}{H \vdash A \to B\ type}$	$\dfrac{H, x{:}A \vdash b\ in\ B}{H \vdash \lambda x.b\ in\ A \to B}$
	$\dfrac{H \vdash A = A',\ H \vdash B = B'}{H \vdash A \to B = A' \to B'}$	$\dfrac{H, x{:}A \vdash b = b'\ in\ B}{H \vdash \lambda x.b = \lambda x.b'\ in\ A \to B}$
elim	$\dfrac{H, B\ type \vdash C\ type}{H, A \to B\ type, A\ type \vdash C\ type}$	$\dfrac{H, a\ in\ A,\ b[a/x]\ in\ B \vdash c\ in\ C}{H, a\ in\ A,\ \lambda x.b\ in\ A \to B \vdash c\ in\ C}$

Figure 3.2. *Intro* and *Elim* Rules for the Function Space Constructor.

instance, resembles the &-introduction rule, and the elimination rules for function types look like modus ponens. In fact, in an expressive enough type theory it is possible to encode both the propositions and the proof theory of the constructive predicate calculus. We shall examine this point in greater detail later, but intuitively the correspondence between propositions and types (the "propositions-as-types" principle due to Curry [Curry, Feys and Craig 58] and Howard [Howard 80]) depends on identifying the constructive notion of provability with the notion of type inhabitation. In the same way that propositions may or may not be provable, types may or may not be inhabited, so that we may think of a proposition as a type whose elements are proofs of the proposition. Proving that a type is inhabited, then, using the rules of the type theory corresponds to proving the corresponding proposition.

3.2.4. Computing in Type Theory and Set Theory

We conclude the discussion of general type theory with some remarks on some of the aspects of type theory and set theory for describing computational mathematics. We believe that type theory represents a more natural forum for discussing and reasoning about computation than does set theory. The primary reasons we feel this way stem from the fact that the notion of computation, of term reduction and manipulation, is explicitly present in type theory and not in set theory, and the fact that type theory implicitly specifies a logic for reasoning about objects as well as classifying objects while set theory does not. The semantics of type theory is inherently computational since they are given in terms of a universe of terms and a computation system on top of these terms, whereas the semantics of set theory generally are not. Constructive logic can also be modeled in a sufficiently expressive type theory in a moderately straightforward fashion, while the notion of logic is external to the axioms of set theory.

3.3. The Nuprl Type Theory

This section describes in some detail the Nuprl type thoery. Our description of this type theory will parallel the general development of type theory given in the previous section; a full presentation of the theory exists in [Constable et al. 86].

3.3.1. Terms and Computation

The description of the theory begins with a description of the collection of Nuprl terms and the computation system that defines the reductions that apply to these terms. To begin with we shall present the canonical and noncanonical terms in the theory that are pertinent to understanding the rest of the paper. The canonical terms are described in table 3.1. It is worth noting than a Nuprl term is considered canonical if the outermost term constructor is canonical; thus any pair of terms, regardless of whether or not the terms themselves are reducible, is itself a canonical term.

Before giving an account of the noncanonical terms we shall describe a couple of notational conventions in Nuprl terms. Variables (denoted in the presentation by $x,y,z,\ldots$) occurring to the left of a dot in an expression *bind* any free occurrences of these variables in the term to the right of the dot. Furthermore, noncanonical terms have *principal* arguments, which are subterms that must be evaluated before the term itself is computed. Table 3.2 describes noncanonical terms.

The remainder of the section describes the rules used to compute the noncanonical terms. In general it is the case that reductions will make sense only if the principal arguments to the noncanonical form evaluate to the appropriate form; for example, $t_1 + t_2$ can be evaluated if and only if t_1 and t_2 both evaluate to integers. The following notation will be used in the remainder of this section. The term $t[t'/x]$ denotes the substitution of term t' for free variable x, with bound variables in t renamed as necessary to avoid the capture of any free variables in t'. Subscripted t's will denote arbitrary

Table 3.1. Canonical Term Constructors in Nuprl.
t, t', t'' represent terms.
x,y represent variables.

Terms	Intended Meanings
0, 1, -1, 2, -2 ...	Integers.
axiom	Member of certain kinds of type that will be described later.
nil	The empty list.
inl(t), inr(t)	Left and right members of disjoint union types.
$\lambda x.b$	Functions.
$<t',t''>$	Pairs.
$t'.t''$	Lists with head t' and tail t''.
void	A type with no elements.
int	The type of integers.
t' *list*	The type of lists of elements from t'.
$t' \mid t''$	The disjoint union of t' and t''.
$x{:}t' \# t''$	Dependent product (also called indexed sum).
$x{:}t' \rightarrow t''$	Dependent function (also called indexed product).
$\{x{:}t' \mid t''\}$	The type of elements of t' "satisfying" t''.
$(x,y){:}t' // t''$	The type of elements of t' with equality relation defined by t''.
rec(x.t)	Recursively defined types.
$t' = t''$ *in* t	Equality types.
$U_1, U_2, \ldots$	Universes.

Table 3.2. Noncanonical Term Constructors in Nuprl.
t, t_1, t_2, t_3, t_4 represent terms.
Underscored terms represent principal arguments.

Term	Description
$-\underline{t}$	Negation.
$\underline{t_1} + \underline{t_2}$	Addition.
$\underline{t_1} - \underline{t_2}$	Subtraction.
$\underline{t_1} * \underline{t_2}$	Multiplication.
$\underline{t_1} / \underline{t_2}$	Integer division.
$\underline{t_1}\ mod\ \underline{t_2}$	Integer modulus function.
$\underline{t_1}(t_2)$	Function application.
$spread(\underline{t_1};x,y.t_2)$	A term constructor for analyzing the components of pairs.
$decide(\underline{t_1};x.t_2;y.t_3)$	A term constructor for analyzing *inl* and *inr* terms.
$list\_ind(\underline{t_1};t_2;x,y,z.t_3)$	A list induction form.
$ind(\underline{t_1};x,y.t_2;t_3;z,w.t_4)$	An integer induction form.
$rec\_ind(\underline{t_1};x,y.t_2)$	A generalized recursive form for recursively defined objects.
$int\_eq(\underline{t_1};\underline{t_2};t_3;t_4)$	An if-then-else form for integer equality.
$less(\underline{t_1};\underline{t_2};t_3;t_4)$	An if-then-else form for integer ordering.

terms, (subscripted) n's will denote arbitrary integers, and x, y, z, .. will represent variables. Redexes will appear to the left of an $\Rightarrow$, and the associated contractum will appear to the right.

- $-n$ $\Rightarrow$ the negation of n.
- $n_1 + n_2$ $\Rightarrow$ the sum of n_1 and n_2.
- $n_1 - n_2$ $\Rightarrow$ the difference of n_1 and n_2.
- $n_1 * n_2$ $\Rightarrow$ the multiplication of n_1 and n_2.
- n_1 / n_2 $\Rightarrow$ the integer n such that $(n-1)*n_2 \leq n*n_2 \leq n_1$.
- $n_1 \mathit{ mod } n_2$ $\Rightarrow$ the result of $n_1 - (n_2*(n_1/n_2))$.

- $(\lambda x.t_1)(t_2) \quad \Rightarrow \quad t_1[t_2/x]$

  β-reduction in the theory works as one would expect.

- $\mathit{spread}(<t_1,t_2>; x,y.t_3) \quad \Rightarrow \quad t_3[t_1/x, t_2/y]$

  The first component of the pair is substituted for x and the second component of the pair is substituted for y in t_3. To illustrate the use of the *spread* form, we may define the standard projection functions in the following fashion.

  $$\mathit{first} = \lambda p.\mathit{spread}(p;x,y.x)$$
  $$\mathit{second} = \lambda p.\mathit{spread}(p;x,y.y)$$

- $\mathit{decide}(\mathit{inl}(t_1); x.t_2; y.t_3) \quad \Rightarrow \quad t_2[t_1/x]$

  $\mathit{decide}(\mathit{inr}(t_1); x.t_2; y.t_3) \quad \Rightarrow \quad t_3[t_1/y]$

  If the first argument to the *decide* term is an *inl* term then the term surrounded by the *inl* is substituted for the free variable in the second argument to the *decide* term. If the first argument is an *inr* term then the term surrounded by the *inr* is substituted for the free variable in the third argument to the *decide* term.

- $list\_ind(nil;\, t_3;\, x,y,z.t_4) \Rightarrow t_3$

  $list\_ind(t_1.t_2;\, t_3;\, x,y,z.t_4) \Rightarrow t_4[t_1/x,\, t_2/y,\, list\_ind(t_2;\, t_3;\, x,y,z.t_4)]$

  If the first argument to a *list_ind* form is an empty list then the second argument, the "base case" of the induction, to the form is returned. Otherwise, if a nonempty list is the first argument to the form then the third argument, the "inductive case," to the form is returned, with the term corresponding to the head of the first argument substituted for the first free variable, the term corresponding to the tail substituted for the second free variable, and the result of evaluating the *list_ind* form on the tail substituted for the third free variable. For example, we may define the standard *head* and *tail* functions on lists as follows.

  $$head = \lambda l.list\_ind(l;\, nil;\, x,y,z.x)$$

  $$tail = \lambda l.list\_ind(l;\, nil;\, x,y,z.y)$$

  The value returned by these functions in the case that l is *nil* can be arbitrary; we chose *nil*. The next function calculates the sum of the elements in a list of integers.

  $$sum = \lambda l.list\_ind(l;0;x,y,z.x+z)$$

  If l is *nil* then 0 is returned; otherwise, the result of adding the value of the head to the sum of the tail is returned.

- $ind(0;\, x,y.t_1;\, t_2;\, z,w.t_3) \Rightarrow t_2$

  $ind(n;\, x,y.t_1;\, t_2;\, z,w.t_3) \Rightarrow t_1[n/x,\, ind(n+1;\, x,y.t_1;\; t_2;\, z,w.t_3)/y]$

  if $n<0$

$$ind(n; x,y.t_1; t_2; z,w.t_3) \quad \Rightarrow \quad t_3[n/z, ind(n-1; x,y.t_1; t_2; z,w.t_3)/w]$$
$$\text{if } n>0$$

If the first argument to the *ind* form is *0* then the third argument to the form is returned. (This is the "base case" to the inductive definition.) If the first argument is negative, the form evaluates to its second argument with the value of the first argument substituted for the first free variable and the value of the *ind* form with the first argument incremented by *1* substituted for the second free variable. (This is the inductive case for negative integers.) If the first argument is positive, then the form evaluates to its fourth argument with the value of the first argument substituted for the first free variable and the value of the *ind* form with the first argument decremented by *1* substituted for the second free variable. (This is the inductive case for positive integers.) Using an *ind* form, the factorial function *fact* may be defined as follows.

$$fact \quad = \lambda n.ind(n;x,y.1;1;z,w.z{*}w)$$

If $n<0$ then the value returned is *1*. If $n=0$ then the value returned is *1*. If $n>0$ then the result of multiplying n by the factorial of $n-1$ is returned.

° $rec\_ind(t_1; x,y.t_2) \quad \Rightarrow \quad t_2[t_1/x, \lambda z.rec\_ind(z; x,y.t_2)/y]$
$$\text{if } t_1 \text{ is canonical}$$

This form allows computation to be performed on recursively defined objects. Intuitively, t_2 will determine whether or not t_1 is a base case of the recursive definition; if it is, then t_2 performs the appropriate calculation. If t_1 instead contains instances of recursive definition then

t_2 determines the values associated with these instances by applying $\lambda z.rec\_ind(z;x,y.t_2)$ to the instances and then calculates the appropriate value. For example, suppose t is a binary tree with integers at its leaves; the definition of t guarantees that leaves are represented as $inl(n)$ for some integer n and that subtrees are represented as $inr(<t',t''>)$ for trees t' and t''. The following *rec_ind* form calculates the sum of t's leaves.

$$\begin{aligned} sum \quad = \quad & \lambda t.rec\_ind(t; \\ & x,func.decide(x;leaf.leaf; \\ & subtrees.spread(subtrees; \\ & left,right.func(left)+func(right))) \end{aligned}$$

If t is a leaf ($inl(i)$ for an integer i) then i is returned; otherwise, t is $inr(<t',t''>)$ for trees t' and t'', and the result returned is the sum of the recursive invocation on t' and t''.

° $int\_eq(n_1; n_2; t_1; t_2) \quad \Rightarrow \quad t_1 \quad$ if n_1, n_2 are integers and $n_1 = n_2$

$int\_eq(n_1; n_2; t_1; t_2) \quad \Rightarrow \quad t_2 \quad$ if n_1, n_2 are integers and $n_1 \neq n_2$

This form corresponds to the more usual programming statement

if $n_1 = n_2$ **then** t_1 **else** t_2 **fi**.

° $less(n_1; n_2; t_1; t_2) \quad \Rightarrow \quad t_1 \quad$ if n_1, n_2 are integers and $n_1 < n_2$

$less(n_1; n_2; t_1; t_2) \quad \Rightarrow \quad t_2 \quad$ if n_1, n_2 are integers and $n_1 \geq n_2$

This form corresponds to the more usual programming statement

if $n_1 < n_2$ **then** t_1 **else** t_2 **fi**.

3.3.2 Types, Members and Equality

The next step in describing the Nuprl type theory involves defining which terms denote types and which terms represent equal types. Also, given a type we must define the terms that denote elements of the type and the equality relation for members of the type.

The organization of this section revolves around the description of the types, in particular the canonical types. For each kind of canonical type we shall define equality between types, the canonical members of the type and equality between canonical members of the type. Noncanonical forms are treated uniformly in the theory; a noncanonical term is a type if it evaluates to a canonical type, and a term evaluates to a member of a type if it evaluates to a canonical member of a type. In each case the value of the noncanonical term is the canonical value to which it evaluates.

3.3.2.1. The Base Types

The base types of Nuprl include *int*, the type of integers, *void*, an empty type, and U_i for i a positive integer. The U_i represent types of *universes*, or collections of types; we shall discuss them later in this section. Type equality for base types is trivial; $int=int$, $void=void$, and $U_i=U_j$ if i and j are equal integers. The members of *int* include the integers, and the equality between members of *int* corresponds to the standard notion of integer equality. The type *void* by definition contains no members.

3.3.2.2. Simple Type Constructors

The type constructors supplied by the theory can be broken into five classes that we shall describe in turn. The *simple* constructors include ones for building disjoint unions, Cartesian products, and function spaces of types and one for building a type of lists of elements of a given type. In the following assume that A, B, A' and B' are types having elements a, b, a' and b', respectively. $A|B$ represents the type that is the disjoint union of A and B; $A|B=A'|B'$ as types if and only if $A=A'$ and $B=B'$. Canonical elements of $A|B$ have either the form *inl(a)* or *inr(b)* and are equal exactly when they have the same outermost constructor (*inl* or *inr*) and subterms equal in the appropriate type (A if the outermost constructor is *inl* and B if the outermost constructor is *inr*). $A\#B$ represents the Cartesian product of A and B, and $A\#B=A'\#B'$ as types if and only if $A=A'$ and $B=B'$. $A\#B$ has as its canonical elements pairs $<a,b>$. Two canonical elements in this type are equal if and only if their first components are equal in A and their second components are equal in B. $A\rightarrow B$ defines the type of functions from A to B, and $A\rightarrow B=A'\rightarrow B'$ if and only if $A=A'$ and $B=B'$. The canonical elements of $A\rightarrow B$ are terms of the form $\lambda x.b$ such that if a is in A then $b[a/x]$ is in B and if $a=a'$ in A then $b[a/x]=b[a'/x]$ in B. Two functions in this type are equal if they return the same answer for each element in A. A *list* denotes the type of lists of elements of A, and A *list*$=A'$ *list* if and only if $A=A'$. Its canonical elements include *nil* and terms of the form $a.l$, where a is in A and l is in A *list*. Two canonical elements of A *list* are equal if they are both *nil* or if they are $a.l$ and $a'.l'$ with $a=a'$ in A and $l=l'$ in A *list*.

3.3.2.3. Dependent Type Constructors

The next two type constructors implement Martin-Lof's Π and Σ types [Martin-Lof 82]; these types correspond to the set-theoretic notions of indexed set product and indexed set sum, as we shall see. In what follows, we shall assume that A and A' are types and that B and B' are expressions, possibly having x as a free variable, such that for each a in A and a' in A', $B[a/x]$ and $B[a'/x]$ are types and for each a_1 and a_2 in A and a_1' and a_2' in A' where $a_1 = a_2$ in A and $a_1' = a_2'$ in A', $B[a_1/x] = B[a_2/x]$ and $B'[a_1'/x] = B'[a_2'/x]$; when this is the case we say that B (B') is *type-functional* with respect to A (A'). In Nuprl, $\Pi x{:}A.B$ is written as $x{:}A \rightarrow B$, and $x{:}A \rightarrow B = x{:}A' \rightarrow B'$ if and only if $A = A'$ and for each a in A, $B[a/x] = B'[a/x]$. Intuitively, $x{:}A \rightarrow B$ should have elements that look like sequences indexed by A, where the a^{th} element of the sequence comes from type $B[a/x]$. These sequences have natural interpretations as functions whose codomain type depends on the value supplied as an argument to the function; that is, sequence s can be thought of as a function from its index type, A, to a family of types represented by B such that for a in A, $s(a)$ has type $B[a/x]$. (We should note here that if x is not free in B then B must be a type, and $\Pi x{:}A.B$ is isomorphic to $A \rightarrow B$.) Therefore, the canonical forms in Nuprl that inhabit $\Pi x{:}A.B$ have the form $\lambda x.b$, where for each a in A, $b[a/x]$ has type $B[a/x]$ and for each a and a' equal in A, $b[a/x] = b[a'/x]$ in $B[a/x]$. Two canonical elements, $\lambda x.b$ and $\lambda x.b'$, in $\Pi x{:}A.B$ are equal exactly when for any a in A, $b[a/x] = b'[a/x]$ in $B[a/x]$.

$\Sigma x{:}A.B$, the indexed sum (or dependent product) constructor, is written in Nuprl as $x{:}A\#B$; $x{:}A\#B = x{:}A'\#B'$ if and only if $A = A'$ and for each a in A, $B[a/x] = B'[a/x]$. Intuitively, elements of this type should consist of a value and a tag of type A signifying from which type in the family of types B the first value comes from. (It should be noted that in the case that x is not free in B, B is a type, and $\Sigma x{:}A.B$ is isomorphic to $A\#B$.) Accordingly, the canonical Nuprl forms inhabiting $\Sigma x{:}A.B$ are pairs $<a,b>$, where a is in A and b is in $B[a/x]$. Two pairs, $<a,b>$ and $<a',b'>$, in this type are equal if and only if $a = a'$ in A and $b = b'$ in $B[a/x]$.

3.3.2.4. Equality Types

The third type kind of constructor we discuss is used to reflect reasoning about equality of elements in types into the type theory and corresponds to Martin-Lof's I type constructor. These types, called *equality types*, have the form $a = b$ *in* A for A a type and a and b elements in A, and if A' is a type with elements a' and b' then ($a = b$ *in* A) = ($a' = b'$ *in* A') exactly when $A = A'$, a and a' are equal elements of A, and b and b' are equal elements of A. The type $a = b$ *in* A is inhabited by a special term, *axiom*, if and only if a and b are equal in A according to the criteria for equality associated with A. Equality within these types is correspondingly trivial.

3.3.2.5. Set Types, Quotient Types and Propositions as Types

Understanding the fourth kind of type constructors requires an understanding of the propositions-as-types principle, so we shall review this principle in the context of Nuprl. The Nuprl type theory is sufficiently expressive to model the constructive predicate calculus with equality. In order to do so one identifies the logical notion of provability with the type-theoretic notion of type inhabitation and then models propositional connectives as type constructors. Under this interpretation the rules for constructing elements in a type correspond precisely with a set of natural-deduction style proof rules for constructive predicate calculus.

What follows constitutes a brief sketch of the encoding of propositions as types. *False* has no proof, and *void* has no elements, so *False* may be identified with *void*. The proposition $A\&B$ is provable exactly when A is provable and B is provable, and $A\#B$ is inhabited exactly when A is inhabited and B is inhabited, so the propositional connective & may be identified with the type constructor #. The proposition $A \vee B$ is provable constructively if one can prove either A or B and say which was proven, while the type $A|B$ is inhabited exactly when either A or B is inhabited, with the form (*inl* or *inr*) of the term in $A|B$ indicating which type, A or B, is inhabited. Therefore, $\vee$ may be modeled by $|$. The proposition $A \Rightarrow B$ has a proof if, under the assumption that A is provable, B is provable, just as the type $A \rightarrow B$ has an element if the inhabitation of A implies the inhabitation of B, so the identification of $\Rightarrow$ with $\rightarrow$ is justified. The proposition $\neg A$ is provable if the assumption of A leads one to conclude *False*, so $\neg A$ may be modeled by $A \rightarrow void$.

Modeling the constructive quantifiers $\forall$ and $\exists$ is somewhat subtle. The proposition $\forall x{:}A.B$ has a proof if for arbitrary a in A, $B[a/x]$ has a proof, while the type $x{:}A \rightarrow B$ has an inhabiting element if for every a in A, $B[a/x]$ is inhabited; therefore, universally quantified propositions correspond to Π types. Similarly, $\exists x{:}A.B$ may be modeled as $x{:}A\#B$, since $\exists x{:}A.B$ is provable constructively exactly when one can demonstrate an element of A satisfying B, while $x{:}A\#B$ is inhabited if there is an element a of A such that $B[a/x]$ has an element in it. The equality types allow statements about equality in constructive predicate calculus to be modeled in the type theory, so the encoding is complete.

With the propositions-as-types principle in hand we turn to a discussion of the fourth kind of type constructors. Let A and A' be types, and let B and B' be type-functional with respect to A and A', respectively, with free variable x. The *subtype* constructor, $\{x{:}A|B\}$, (often referred to as the *set type* constructor because of its syntactic similarity with the classical representation of sets) allows the definition of subtypes of A with respect to B interpreted as a predicate over A. $\{x{:}A|B\} = \{x{:}A'|B'\}$ if and only if $A = A'$ and for each a in A, $B[a/x]$ is inhabited exactly when $B'[a/x]$ is. The type $\{x{:}A|B\}$ has as its canonical elements the elements a of A such that $B[a/x]$ interpreted as a proposition is true. Two elements in $\{x{:}A|B\}$ are equal if and only if they are equal in A.

The *quotient* type constructor allows equality to be redefined for a type. Let A and A' be types, and let B and B' be expressions having x and y free such that B (B') is type-functional with respect to A (A') in both x and y and such that B and B' are equivalence relations with respect to A and A', respectively. Then $(x,y):A//B$ redefines the equality in type A as B. $(x,y):A//B = (x,y):A'//B'$ exactly when $A = A'$ and for all a and a' in A, $B[a/x, a'/y]$ is inhabited if and only if $B'[a/x, a'/y]$ is. The canonical elements of $(x,y):A//B$ are the canonical elements of A; two elements a and a' in $(x,y):A//B$ are equal exactly when $B[a/x, a'/y]$ is inhabited.

3.3.2.6. Recursive Types

The final type constructor we shall cover allows the definition of *recursive* types. The type of binary trees having integers at their leaves, for example, may be seen as the type of closed finitary "solutions" to the equation $z = int \mid z \times z$; in Nuprl the type is written $rec(z.int|z \times z)$. Figure 3.3 gives examples of trees in this representation.

In general recursive types in Nuprl have the form $rec(z.T)$, where T is an expression possibly having z as a free variable and satisfying the following conditions.

(1) The variable z may not occur in the domain type of a function space or Π type, in the argument of a function application, or in the principal arguments of noncanonical forms.

(2) For all types S in some universe U_j, $T[S/x]$ is a type.

Given two expressions T and T' satisfying the previous two conditions, $rec(z.T) = rec(z.T')$ exactly when for any type S in some universe U_j, $T[S/z] = T'[S/z']$. The canonical terms inhabiting recursive types are the

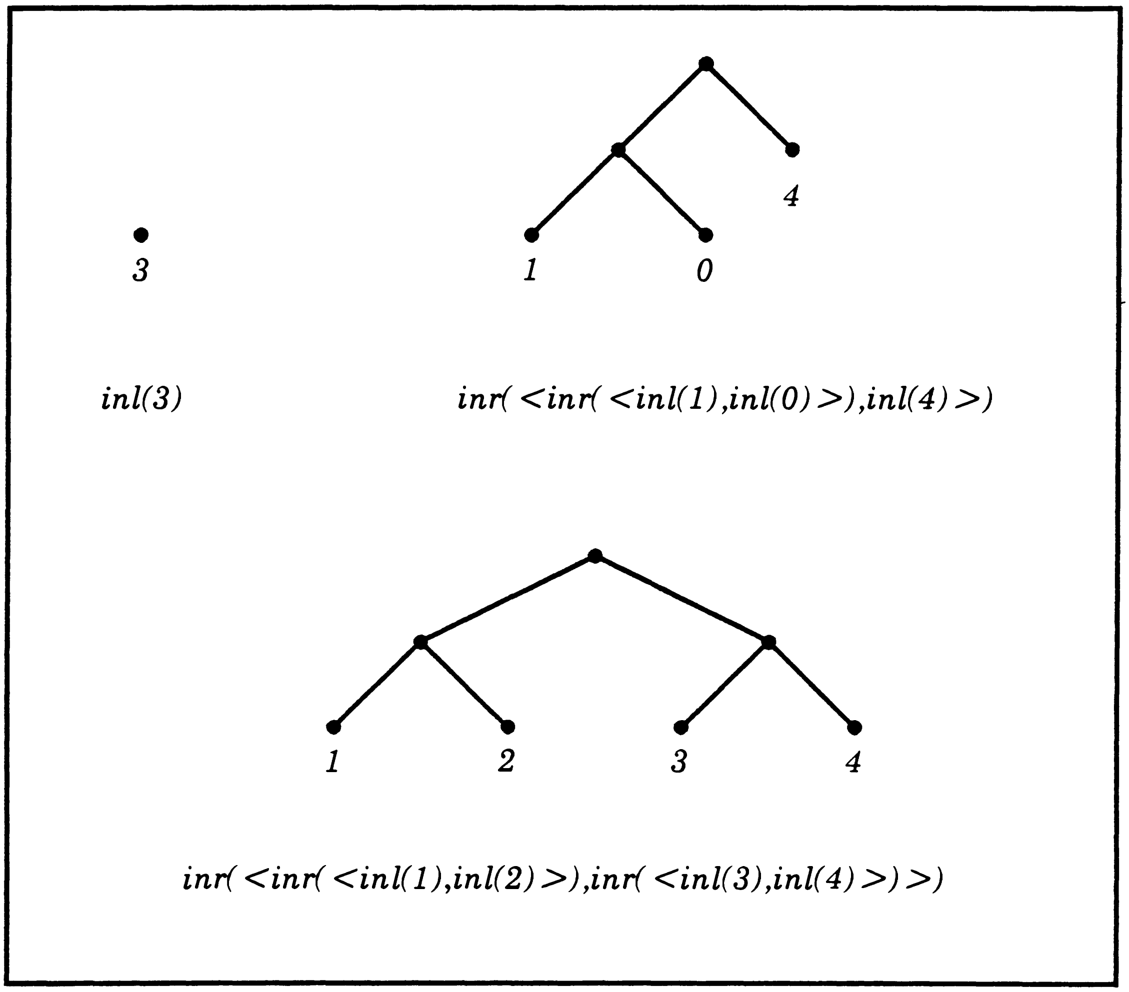


Figure 3.3. Examples of Binary Trees.

canonical forms associated with the outermost type constructor in T; in the tree example, the canonical terms are *inl* and *inr* terms. Equality between terms in recursive types is the equality associated with the type constructors in T.

3.3.2.7. Universes

We close our description of the Nuprl type system with a discussion of universes. The desirability of polymorphic computation and type-valued functions has been noted in much of the literature on types, and allowing the possibility of these kinds of objects requires that collections of types themeselves be treated as types. Several approaches to doing so have been suggested, some of which have led to inconsistent systems [Martin-Lof 71] and some of which have led to impredicative systems that are not known to be consistent [Coquand and Huet 85]. The Nuprl type system addresses the issue in a consistent and predicative fashion by providing a cumulative hierarchy of type *universes* $U_1, U_2, \ldots$. These universes are defined in such a way that the following properties hold.

(1) The types *int* and *void* are in U_1.

(2) If T_1 and T_2 are in U_i then so are any types built from T_1 and T_2 using the type constructors described above.

(3) If T is in U_i then T is in U_{i+1}.

(4) U_i is in U_{i+1}.

The fact that types are themselves members of types dictates that type equality corresponds to the equality between members of universe types.

3.3.3. Rules and Proof Theory

The logic associated with Nuprl is somewhat different from traditional natural deduction logics in that the Nuprl logic is a *refinement*, or top-down, logic. That is, one proves goals in the logic by *refining* them using the rules of the logic; applying a rule to a goal results in a number of subgoals that must be proved in order to establish the main goal. Nuprl rule descriptions, then, are upside-down versions of traditional natural deduction-style rules, with the conclusion of the rule stated first and the subgoals stated underneath the conclusion. For example, the #-intro rule in Nuprl looks like this.

$$\frac{H \vdash A \# B}{\begin{array}{c} H \vdash A \\ H \vdash B \end{array}}$$

Each type and type constructor has associated with it a number of intro rules and elim rules, with intro rules essentially specifying when a type may appear as a goal and elim rules specifying how types that appear as hypotheses may be broken down and analyzed. Furthermore, types have rules detailing when they are well-formed (that is, when they reside in a universe) and rules allowing equality between elements of types to be proven; these rules are also called intro rules. The logic also has some miscellaneous rules, such as a hypothesis rule (allowing the conclusion of a goal that also appears as a hypothesis), a cut rule (allowing one to introduce facts into the hypothesis list, as long as one proves the facts), a lemma rule and a special rule for performing certain kinds of arithmetic reasoning.

The construction rules for types correspond exactly to proof rules of the constructive predicate calculus for the associated propositions. We shall in fact refer to type rules by their propositional counterparts on occasion. Thus, for instance, we shall say $\forall$-intro instead of Π-intro and and-intro instead of Σ-intro.

3.3.4. Proofs and Programs

While we have alluded to the correspondence between proofs and programs in Nuprl we have not described the mechanism that implements this correspondence. The semantics of propositions in Nuprl dictate that propositions are provable exactly when a term can be constructed that inhabits the type corresponding to the proposition. Associated with each proof rule, then, is a term template whose arguments are filled by the terms specified by proofs of the subgoals generated by the invocation of the proof rule. For example, proving $\vdash A\#B$ by #-intro implicitly specifies a term that is a pair $<a,b>$, where a is specified by the proof of A and b by the proof of B. The process of constructing a term from a proof in Nuprl is called *term extraction*. It is beyond the scope of this thesis to describe term extraction in detail, but in general intro rules give rise to canonical terms while elim rules give rise to noncanonical forms. In order to extract the term that a proof specifies, the theory has a special term constructor, *term_of*, that takes as its only argument the name of a theorem and produces the term corresponding to the proof of the theorem.

3.4. The Nuprl System

The Nuprl type theory described above forms the theoretical basis for the Nuprl system. This system supports the interactive development of proofs in the type theory and the extraction of computable terms from proofs of theorems. The system also features a very flexible mechanism for user-defined notation, a metalanguage--the programming language ML--in which derived rules of inference, known as *tactics*, [CKB84] may be written, an evaluator that computes the terms of the theory, and a library facility for storing theorems, tactics, notation extensions (hereafter called *defs*) and objects for the evaluator. In this section we shall describe some of the features of the Nuprl system in more detail; the full system description appears in [Constable et al. 86].

Interaction with the system occurs in the context of special windows on the terminal screen. Nuprl has four kinds of windows, each of which serves a specific function, and it supports a variety of commands for manipulating windows. What follows constitutes a brief description of each type of window. The *command* window is the window from which all interaction with the system commences. In this window one can enter commands that cause objects to be created in, deleted from, or moved within the library, commands that allow one to edit objects in the library, commands that invoke the evaluator or the ML interpreter, and commands that cause the loading and storing of the library from the file system of the underlying operating system. The *library* window contains a list of theorems, defs, tactics and evaluation objects that have been created or loaded in the current Nuprl session. *Refinement editor* (or *red*) windows appear when one edits a theorem in the library and allow the development of proofs and the viewing of proofs that have been developed. *Text editor* (or *ted*) windows appear when one edits def, tactic or evaluation objects in the library or when one enters rules and theorem statements in the context of a red window; these windows allow the manipulation of text.

Because of the refinement-style logic that underlies the Nuprl theory the Nuprl system supports the top-down development of proofs. The system represents proofs as trees of nodes, where a node contains a hypothesis list, a goal, a rule (or tactic) invocation, and a list of subgoals generated automatically by red on the basis of the form of the goal and the rule (or tactic) used. Assuming the consistency of the Nuprl logic, the automatic generation of subgoals by red ensures the correctness of proofs developed in the system; in this sense, Nuprl acts as an interactive proof checker. To state and prove a theorem, one first creates a theorem object in the library and edits it by invoking red. One enters the theorem statement by editing a special slot in the red window; the system responds with a ted window, the user enters the theorem statement and closes the ted window, and the Nuprl parser checks the syntactic well-formedness of the statement. Assuming the statement of the theorem is valid, one is now ready to prove the theorem. To do so, one edits the rule slot in the current node; the system responds with a ted window, into which one types the name of the rule or tactic one wishes to apply. After the user closes the ted window, the parser checks the syntax of the object typed in, and red determines the applicability of the given rule or tactic and, if it is applicable, calculates the subgoals that must be proven in order to achieve a proof of the main goal. If no children result from the refinement then the proof is complete; otherwise, one uses red to descend into the child nodes of the current node and prove the corresponding subgoals.

We conclude this section with a more detailed description of the role of tactics in the proof development process. Tactics are functions written in a version of the ML programming language that includes types corresponding to the components of Nuprl proof trees--rules, hypothesis lists, goals, subgoals, and the like--and intuitively serve as derived rules of inference. The consistency of these tactics is guaranteed by the fact that these functions essentially apply the inference rules associated with the Nuprl logic in ways specified by the ML source; thus a tactic invocation corresponds to a string of applications of Nuprl inference rules. One may enter a tactic name anywhere that one may enter a rule name; the system automatically ascertains whether a rule or a tactic is being called for, and in the latter case the ML interpreter, running in conjunction with red, attempts to construct the proof the tactic calls for.

Chapter 4

An Acceptance Model of Concurrency

This chapter defines a model of concurrent processes. The model characterizes processes in terms of their traces and the choices available at the end of each trace. The model is somewhat similar to the failures model of CSP described in chapter 2 and the testing model described in chapter 2 and even more so to the acceptance model due to Hennessy [Hennessy 83]; we shall examine the simlarities in more detail later in the chapter. However, the process equivalence induced by the model we are about to present is finer than the equivalence in either of the first two models, and the simple trace-based description that objects in the model have render it a suitable model of temporal logic. The simplicity of the description also make it relatively easy to implement in Nuprl.

The equality in the model described in this chapter turns out to be strictly coarser than the strong equivalence of CCS and incomparable with observational equivalence in CCS; both equivalences are defined in chapter 2. While these facts make it impossible to give a full account of CCS in this model, the model does have interesting characteristics. The history-based nature of the representation of processes enables a natural account of temporal logic to be given; furthermore, the description of the choices available to a process at each point in its history enable an interesting range of temporal properties to be described. It has also been argued [Strom 86] that the equivalences in CCS make inappropriate distinctions between processes. This argument will be presented in more detail later in the chapter; it suffices now to say that the equivalence induced by this acceptance model distinguishes only those processes which may be distinguished on the basis of sequences of single experiments.

The chapter is organized along the following lines. The first section describes the model in set-theoretic terms, and the second section presents a semantic account of the program constructors of CCS, SCCS and CSP. The third section characterizes the equivalence between processes induced by equality in the model, the fourth describes how the model may be used to give a semantic account of temporal logic, and the last section discusses implementing the model in Nuprl.

4.1. A Set-Theoretic Account of the Model

The intuition for this model stems from the fact that in the course of its execution a process will at any point have a selection of possible actions that it may perform next. Accordingly, this model characterizes a process in terms of the possible initial courses of execution (traces) and the choices of action available to a process after pursuing a course of execution.

More formally, let A be the set of *atomic actions* that processes may perform, let A^* be the set of finite sequences (or traces) of elements of A, and let *finset(A)* denote the collection of finite subsets of A. If p is a pair then let $p.1$ and $p.2$ denote the first and second components of the pair; if $l \in A^*$, $l' \in A^*$ and $a \in A$ then let $l\hat{}a$ be the result of appending a onto l, $a\hat{}l$ be the result of prepending a onto l, and $l\hat{}l'$ be the result of concatenating l and l'. A process P is a subset of $A^* \times$ *finset(A)* satisfying the following conditions.

(1) There is a $p \in P$ such that $p.1 = \varepsilon$.

(2) If $p \in P$ and $a \in p.2$ then there is a $q \in P$ such that $q.1 = p.1\hat{}a$.

(3) If $p \in P$ and $p.1 = l\hat{}a$ for some $a \in A$ then there is a $q \in P$ with $q.1 = l$ and $a \in q.2$.

Let *Proc* denote the collection of subsets of $A^* \times$ *finset(A)* satisfying conditions (1)-(3). *Proc* is the model of computation that will be discussed in the rest of this chapter.

Processes are interpreted in *Proc* as (potentially infinite) sets of pairs consisting of a trace and a *next set*, which is a finite set of atomic actions that the process, after performing the associated trace, may next choose to perform. The three conditions impose a set of consistency restraints on the subsets of pairs that may be interpreted as processes. Condition (1) requires that each process have a starting point. Condition (2) states that if at a certain point in its execution an action is available to a process then it must be possible for the process to continue executing by performing that action. Condition (3) requires that a process can reach a certain point in its execution only by executing actions that are possible at each step in the execution of the process leading up to the point in question.

One aspect of this model that deserves mention is the lack of a functionality requirement between traces and next sets. In general a trace in a process may have many next sets associated with it in the sense that distinct pairs in the representation of a process may have the same trace component. The intended interpretation of this is that as a result of nondeterministic choices made during the course of its execution a process may have performed the same sequence of actions but have different sets of actions available to it.

Figure 4.1 gives examples of processes represented in *Proc*. In the figure processes are represented as synchronization trees (see chapter 2), where the root denotes the starting point of a process, labels on arcs represent actions, branching denotes choice and paths denote traces. These examples suggest several remarks regarding this theory. Terminated computations are represented by traces having empty next sets associated with them. Also, different kinds of choice structures may be represented distinctly.

This model bears comparison with other similar models of concurrency that have been proposed. The failures model of CSP due to Brookes, Hoare and Roscoe [Brookes, Hoare and Roscoe 84] is syntactically similar to *Proc* in that processes are represented as sets of trace-finite set pairs; however, the finite sets of actions in the failures model denote actions which a process

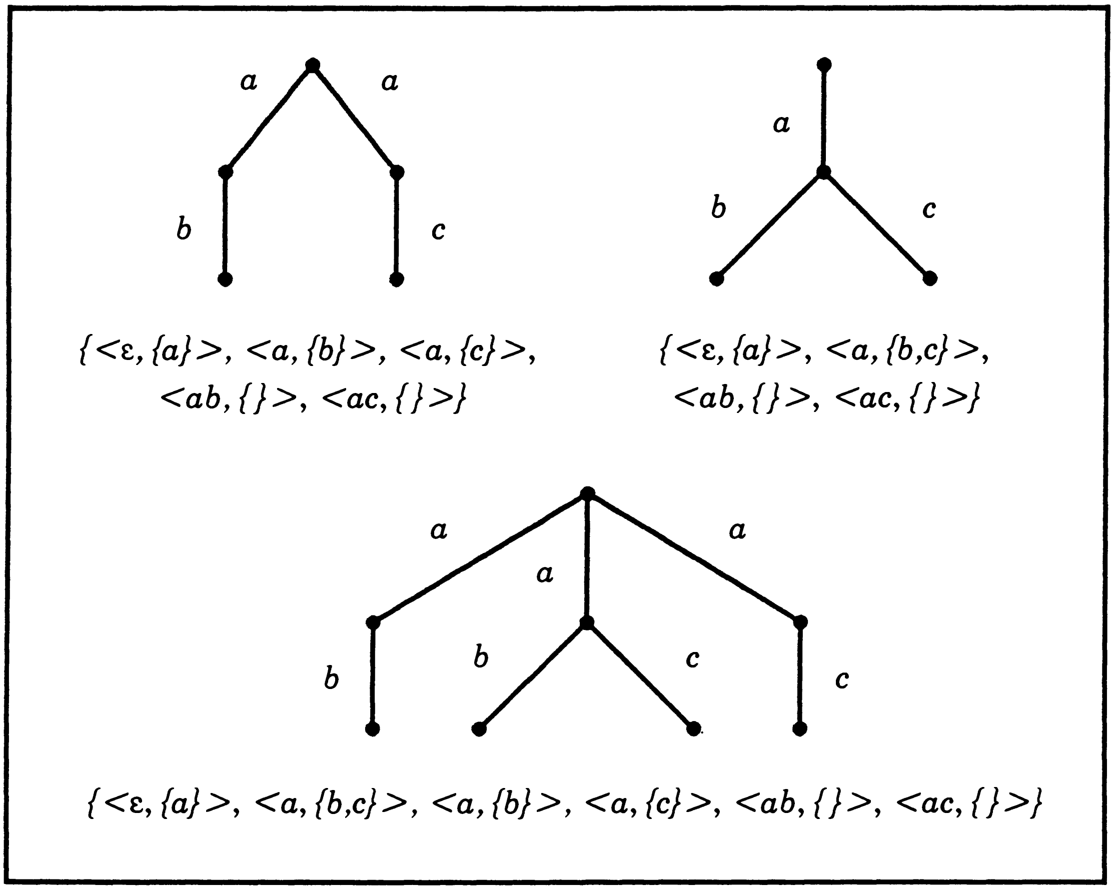


Figure 4.1. Examples of Synchronization Trees as Elements of *Proc*. a, b, c are elements of A.

may *fail* to perform after executing the associated trace. While a particular trace may have many different so-called *refusal sets* associated with it, there is a *maximal* set of actions corresponding to the trace that the process may refuse to perform; if the set of actions is finite, this set will also be a refusal set.

The version of the deNicola-Hennessy testing model of processes [deNicola and Hennessy 82, Hennessy 82] mentioned in chapter 2 characterizes processes in terms of sequences of states the process may enter, the set of actions a process *may* perform after a sequence, and the set of actions a process *must* perform after executing a sequence. Both sets are assumed finite.

The acceptance model proposed by Hennessy [Hennessy 83] is more similar to *Proc* than the previous two models in that possible next actions are recorded in the representation of processes. The mathematical machinery needed to express this model is somewhat complicated. Processes are represented by trees with arcs labeled by actions and certain nodes labeled by sets of sets of actions; these sets (of sets) are called *acceptance sets*. The arcs leaving a node labeled by an acceptance set must be labeled by an element in one of the elements in the acceptance set. Trees have the additional property that any node can have at most one arc labeled by a particular action leaving it, and elements of acceptance sets satisfy several conditions, including finiteness (both of themselves and of their member sets), convex closure and closure with respect to union. An unlabeled node corresponds to an undefined state in the process; only unlabeled nodes may have an infinite number of arcs leaving them. Labeled nodes denote completely defined sections of processes, and these nodes may only have a finite number of branches emanating from them. As in the *Proc* model, acceptance sets describe the actions that a process may perform as a result of nondeterministic choices made in the course of its computation. The restrictions placed on acceptance sets enable the acceptance model to be treated as a complete partial order under an "is less deterministic than" ordering, thereby allowing the machinery in the well developed area of fix-point semantics to be used in analyzing the model and using it to describe the denotational semantics of programming languages.

The following terms will appear throughout the rest of the chapter. For an element P of *Proc*, the *traces* of P are the elements of the set $\{l \in A^* \mid \exists s \in finset(A).\ <l,s> \in P\}$. The next sets *associated* with a trace l are elements of the set $\{s \in finset(A) \mid <l,s> \in P\}$.

4.2. Developing CCS, SCCS, and CSP in the Model

This section describes the semantics of the various CCS, SCCS and CSP term constructors as operations in the *Proc* model. In each case the restraints placed by the concurrency formalism being modeled on A are described; each operator is then developed and the correctness of the derivation argued.

4.2.1. CCS

The CCS formalism imposes a structure on the set A of allowable actions. There must be a distinguished action, $\tau \in A$, that is interpreted as an internal or silent action. Furthermore, if an action $a \in A$ and $a \neq \tau$ then there must be an action $\underline{a} \in A$, with $\underline{\underline{a}} = a$. Intuitively, a and $\underline{a}$ may synchronize; τ may not synchronize with any actions.

In order to argue for the correctness of the following interpretations of CCS process constructors as functions on *Proc* it is necessary to define the CCS transition relation $P \rightarrow^{a} P'$ in terms of *Proc*. Assuming that for a process P, $T(P)$ is the representation of P in *Proc*, the translation of $\rightarrow^{a}$ is:

$$T(P) \rightarrow^{a} T(P') \text{ if and only if } \forall p \in T(P').<a\hat{\ }p.1, p.2> \in T(P).$$

It is not necessary to require that a belongs to a next set associated with ε in $T(P)$, for condition (3) implies that this will be so if the above condition is met. The semantic accounts of the operators follow.

4.2.1.1. NIL

In CCS *NIL* represents the process which does nothing except terminate. In the *Proc* model,

$$T(NIL) = \{ <\varepsilon, \varnothing> \}.$$

It is clearly the case that $T(NIL)$ is an element of *Proc* and that $T(NIL)$ respects the semantics of *NIL*.

4.2.1.2. Prepend

Given a process P we may represent the result of prepending an action a to the beginning of P as follows:

$$T(aP) = \{ <\varepsilon, \{a\}> \} \cup \{ <a\hat{\ }s, N> \mid <s, N> \in T(P) \}.$$

As $T(aP)$ is clearly a member of *Proc*, arguing for the correctness of this account requires showing that $T(aP) \rightarrow^{a} T(P)$. This follows from the definition of the transition relation on elements of *Proc* given above.

4.2.1.3. Restriction

In CCS one may provide for the prevention of actions in a process; $P\backslash a$ corresponds to the process P with all courses of action headed by the action a deleted. The *Proc* representation of this construction is as follows.

$$P\backslash a = \{<s, N> \mid <s, M> \in T(P) \text{ and } s \text{ contains no occurrences of } a \text{ and } N = M - \{a\}\}$$

Assuming that $T(P) \in Proc$, we must show that $T(P\backslash a) \in Proc$. Since $<\varepsilon, N> \in T(P)$ for some next set N it follows that $<\varepsilon, N-\{a\}> \in T(P\backslash a)$, and condition (1) is met. Similarly, if $p \in T(P\backslash a)$ and $b \in p.2$ then $b \neq a$, and therefore $p.1\hat{\ }b$ contains no occurrences of a and hence, since $p.1\hat{\ }b \in T(P)$, $p.1\hat{\ }b \in T(P\backslash a)$. Therefore condition (2) is met. A similar argument establishes that condition (3) is satisfied, and $T(P\backslash a) \in Proc$.

The semantics of $P\backslash a$ dictate that $P\backslash a \rightarrow^{b} P'\backslash a$ if $P \rightarrow^{b} P'$ and $b \neq a$, so it remains to be shown that $T(P\backslash a) \rightarrow^{b} T(P'\backslash a)$ assuming that $T(P) \rightarrow^{b} T(P')$ and $b \neq a$. The result follows by a trivial induction on P.

4.2.1.4. Relabeling

Relabeling provides a mechanism for changing the action set of a process. A *relabeling* from action set A to action set B refers to a function $S{:}A \rightarrow B$ such that $S(\tau) = \tau$ and such that $\underline{S(a)} = S(\underline{a})$. We can extend S to map strings in A^* to strings in B^* and finite sets of A to finite sets of B in the natural way. Therefore, given a process P and a relabeling S, define the relabeling of P via S as follows.

$$T(P[S]) = \{ <s,N> \mid \exists p \in T(P)\ s = S(p.1)\ and\ N = S(p.2) \}.$$

The correctness of this follows by induction on P.

4.2.1.5. Nondeterministic Choice

$P+Q$: $P+Q$ represents a process which may behave like either P or Q, with the additional stipulation that the first actions of both P and Q are available to $P+Q$ at the start of its execution. This process constructor may be defined as follows.

$$\begin{aligned} T(P+Q) = \quad & \{<\varepsilon,M'> \mid \quad \exists M,N.\ <\varepsilon,M> \in T(P) \wedge <\varepsilon,N> \in T(Q) \wedge \\ & \qquad\qquad\qquad M' = M \cup N\} \\ & \cup \{<s,M> \mid s \neq \varepsilon \wedge <s,M> \in T(P)\} \\ & \cup \{<s,N> \mid s \neq \varepsilon \wedge <s,N> \in T(Q)\} \end{aligned}$$

Since $T(P+Q)$ satisfies conditions (1)-(3) above it is a member of *Proc*. An argument for the correctness of $T(P+Q)$ must establish that if $T(P) \rightarrow^{a} T(P')$

then $T(P+Q) \rightarrow^a T(P')$ and if $T(Q) \rightarrow^a T(Q')$ then $T(P+Q) \rightarrow^a T(Q')$. These facts follow from the definition of the CCS semantic relation in the model.

4.2.1.6. Concurrent Composition

In CCS the composite process $P|Q$ behaves like the interleaved concurrent composition of P and Q. Defining $T(P|Q)$ is somewhat tricky. Intuitively, the traces of $T(P|Q)$ will include all the possible interleavings of traces of $T(P)$ and $T(Q)$ and the traces resulting from possible synchronizations of inverse actions within the interleavings, where τ denotes the result of the synchronization; call this collection the "synchronized interleavings" of traces of $T(P)$ and $T(Q)$. Determining which synchronizations are possible in an interleaving of two traces requires determining which constituent trace in an interleaving an action comes from. Accordingly, given l_1, $l_2 \in A^*$ the following characterizes the set of "tagged" interleavings of l_1 and l_2.

$shuffle(l_1, l_2) : A^* \times A^* \rightarrow (A \times \{1,2\})^* =$

if $l_1 = \varepsilon \wedge l_2 = \varepsilon$ **then** ε

else

$\{l \in (A \times \{1,2\})^* \mid$

$head(l).1 = head(l_1) \wedge head(l).2 = 1 \wedge tail(l) \in shuffle(tail(l_1), l_2)) \vee$

$head(l).1 = head(l_2) \wedge head(l).2 = 2 \wedge tail(l) \in shuffle(l_1, tail(l_2))\}$

That is, given a trace l_1 and a trace l_2 *shuffle* constructs the set of interleavings of l_1 and l_2 with actions in each interleaving tagged with 1 if they come from l_1 and 2 if they come from l_2.

It will be necessary to strip away the trace tag at some point in order to leave synchronized interleavings; the following function does this.

$strip(l) : (A \times \{1,2\})^* \rightarrow A^* =$

if $l = \varepsilon$ **then** ε

else $head(l).1 \hat{\ } strip(tail(l))$

Now suppose that s_1 and s_2 are traces of $T(P)$ and $T(Q)$, respectively. The following sequence of equations defines a finite collection, $\{S_i\}$, of sets of lists of pairs, where the first components of the pairs form synchronized interleavings of s_1 and s_2. It should be noted that the traces in S_i contain exactly i synchronizations.

$S_0 = shuffle(s_1, s_2).$

$S_{i+1} = \{x \hat{\ } <\tau, 1> \hat{\ } y \mid \exists a \in A. x \hat{\ } <a, i> \hat{\ } <\underline{a}, j> \hat{\ } y \in S_i \wedge i \neq j\}$

Now define the set of synchronized interleavings of s_1 and s_2 as

$Comp(s_1, s_2) = strip(\cup S_i)$

and the set of actions available to the composition of two processes having M and N as next sets as

$$
\begin{aligned}
Synch(M,N) &= M \cup N \cup \{\tau\} && \text{if } a \in M \text{ and } \underline{a} \in N \text{ some } a \\
&= M \cup N && \text{otherwise.}
\end{aligned}
$$

$P|Q$ may now be defined in the following fashion.

$$
\begin{aligned}
T(P|Q) = \{<s, M'> \mid\ & \exists p \in T(P).\ \exists q \in T(Q).\ s \in Comp(p.1, q.1) \wedge \\
& M' = Synch(p.2, q.2)\}
\end{aligned}
$$

Proving that $T(P|Q) \in Proc$ if both $T(P)$ and $T(Q)$ are requires establishing that conditions (1)-(3) hold. Clearly ε is a trace of $T(P|Q)$, so condition 1 holds. Now suppose that $r \in T(P|Q)$ and $a \in r.2$; showing that condition 2 holds requires establishing that there is a $s \in T(P|Q)$ with $s.1 = r.1\hat{\ }a$. Since $r \in T(P|Q)$ there are p and q, members of $T(P)$ and $T(Q)$, respectively, such that $r.1 \in Comp(p.1, q.1)$ and $r.2 = Synch(p.2, q.2)$. If $a = \tau$ then there is a b with $b \in p.2$ and $\underline{b} \in q.2$, and it is therefore the case that there are p' and q', members of $T(P)$ and $T(Q)$, respectively, such that $p'.1 = p.1\hat{\ }b$ and $q'.1 = q.1\hat{\ }\underline{b}$. By definition, there is an $s \in T(P|Q)$ such that $s = r\hat{\ }\tau$, and we are done. The argument for $a \neq \tau$ proceeds similarly and so is omitted, as is the argument that condition (3) is met.

Establishing the semantic correctness of this implementation requires showing that it meets three semantic conditions. The first two, $T(P) \rightarrow^a T(P') \Rightarrow T(P|Q) \rightarrow^a T(P'|Q)$ and $T(Q) \rightarrow^a T(Q') \Rightarrow T(P|Q) \rightarrow^a T(P|Q')$ follow from the definition of the semantic relation in *Proc*. The third, $T(P) \rightarrow^a T(P') \wedge T(Q) \rightarrow^{\underline{a}} T(Q') \Rightarrow T(P|Q) \rightarrow^\tau T(P'|Q')$, also follows from this.

4.2.1.7. Recursion

CCS allows the definition of mutually recursive behavior expressions. We shall concentrate on modeling the simpler nonmutual recursion operator $fix(x.b)$. Intuitively, $fix(x.b)$ represents a term solving the equation $x = b$. It can be modeled by essentially "unwinding" the recursive expression in the following fashion.

First, it is convenient to extend the translation function T to a function T' that handles free variables. Define $T'(x) = <x, \{\}>$ for x a free variable and $T'(c)$, for CCS expressions c potentially having free variables, as $T(c)$ with $T'(x)$ being returned whenever a free varible x is encountered. Now define the following family of relations.

$$R_0 = T'(b)$$

$$\begin{aligned} R_{i+1} = \; & \{<s,N> \mid \; (<s,N> \in R_i \wedge s \textit{ has no occurrences of } x) \vee \\ & \qquad (\exists p \in R_i, q \in R_0, r.\ p.1 = r\hat{\ }x \wedge s = r\hat{\ }q.1 \wedge N = q.2)\} \end{aligned}$$

Thus R_i represents the (up to) i^{th} unrolling of b. Now define $T(fix(x.b))$ as

$$T(fix(x.b)) = \{<s,N> \mid \exists i.\ <s,N> \in R_i \wedge s \textit{ has no occurrences of } x\}.$$

4.2.2. SCCS

SCCS is a generalized form of CCS that enables the modeling of systems in which processes execute in lock-step. The term constructors, with the exception of a summation operator and the concurrent composition operator, are very similar to those found in CCS. The summation operator in SCCS allows the development of infinitely nondeterministic processes, and as the *Proc* model can only model finite nondeterminism we will be unable to describe this operator in *Proc* (although we can account for finitary versions of it). The concurrent composition operator takes two terms and forces them to execute synchronously; this operator can be modeled.

The set A of atomic actions allowable in SCCS must be an abelian group, where the group operation applied to two elements of A is interpreted as the action resulting from the synchronized execution of the two actions. In what follows $\circ$ denotes the group operation on A.

To define the SCCS concurrent composition operator the semantic relation $\rightarrow^a$ is defined on *Proc* as before. The definition of $P \otimes Q$ is made easier by the following definitions. Let $s_1, s_2 \in A^*$. The function $Glue(s_1,s_2)$ computes the trace corresponding to the synchronous simultaneous execution of s_1 and s_2.

$$
\begin{aligned}
Glue(s_1,s_2) &= \varepsilon && \text{if } s_1 = \varepsilon \text{ or } s_2 = \varepsilon \\
&= (head(s_1) \circ head(s_2))\hat{\ }Glue(tail(s_1), tail(s_2)) && \text{otherwise}
\end{aligned}
$$

Now let $M, N \in finset(A)$; the group operation on A is extended to sets of A by saying that $M \circ N = \{c \mid \exists a \in M, b \in N. c = a \circ b\}$. With this definition we can define $P \otimes Q$ as follows.

$$
T(P \otimes Q) = \{<s, M> \mid \exists p \in T(P).\ \exists q \in T(Q).\ s = Glue(p.1, q.1) \wedge M = p.2 \circ q.2\}
$$

To show this correct it suffices to note that $T(P) \rightarrow^a T(P') \wedge T(P) \rightarrow^b T(P') \Rightarrow T(P \otimes Q) \rightarrow^{a \circ b} T(P' \otimes Q')$.

4.2.3. CSP

The account of CSP due to Brookes, Hoare and Roscoe [Brookes, Hoare and Roscoe 84] specifies a different set of term constructors than CCS, although the philosophy of describing processes in terms of their atomic actions and the semantics of term constructors in terms transition relations is similar. The set A of atomic actions for CSP has no conditions imposed on it, however. The rest of the section describes some of the CSP term constructors that do not appear in CCS.

Interleaving: $P|||Q$ represents the process which can perform any interleaving of events from P and Q. Given the definitions of *shuffle* and *strip* above define $T(P|||Q)$ in the following way.

$$T(P|||Q) = \{<s,M>: \exists p \in T(P).\ \exists q \in T(Q).\ s \in strip(shuffle(p.1,q.1)) \wedge M = p.2 \cup q.2\}$$

Parallel Composition: In CSP $P||Q$ represents the process which corresponds to the simultaneous execution of P and Q, provided P and Q are performing the same actions. We define $T(P||Q)$ as follows.

$$T(P||Q) = \{<s,M> \mid \exists p \in T(P).\ \exists q \in T(Q).\ s = p.1 = q.1 \wedge M = p.2 \cap q.2\}$$

Indeterminate choice: $P + Q$ represents a process which may behave like P or Q. It differs from the CCS $P+Q$ in that the environment has no say as to which process will be simulated. We define this as

$$T(P \oplus Q) = T(P) \cup T(Q).$$

4.3. Analyzing the Equivalence in *Proc*

This section examines the relationship between equality in *Proc* and various behavioral equivalences that have been suggested in other work. The main result of this analysis is that $=_{Proc}$ is strictly coarser than CCS strong equivalence, implying that this model is incomplete as a model of CCS, since in CCS equational reasoning on the basis of process equivalence is central to the theory. However, the equivalence is at least as fine as failures equivalence [Brookes, Hoare and Roscoe 84] and testing equivalence [deNicola and Hennessy 82, Hennessy 82], and this suggests that $=_{Proc}$ is nonetheless an interesting equivalence.

4.3.1. The Relationship with CCS equivalences

The CCS notions of strong and behavioral equivalence were described in chapter 2; for convenience they are restated here. In what follows, let P and Q be arbitrary CCS terms. *Strong equivalence* is defined as the intersection of the iteratively defined family of relations presented below.

$P \sim_0 Q$.

$P \sim_{i+1} Q$ for $i \geq 0$ if and only if:

(i) $P \rightarrow^a P'$ implies that for some Q', $Q \rightarrow^a Q'$ and $P' \sim_i Q'$;

(ii) $Q \rightarrow^a Q'$ implies that for some P', $P \rightarrow^a P'$ and $P' \sim_i Q'$.

Then $P \sim Q$ ("P and Q are strongly equivalent") exactly when $\forall i \geq 0. P \sim_i Q$.

Defining o*bservational equivalence* requires the following notions. For $l \in A^*$ defined $P \rightarrow^l P'$ as being true when either $l = \varepsilon$ and P and P' are identical terms or when there is a Q such that $P \rightarrow^{head(l)} Q$ and $Q \rightarrow^{tail(l)} P'$. Let $s \in (A - \{\tau\})^*$, and let $P \Rightarrow^s P'$ exactly when there exists and $s' \in A^*$ such that $s = s'$ with τ actions removed and $P \rightarrow^{s'} P'$. Observational equivalence is the intersection of the following iteratively defined relations.

$P \approx_0 Q$.

$P \approx_{i+1} Q$ for $i \geq 0$ if and only if:

(i) $P \Rightarrow^s P'$ implies that for some Q', $Q \Rightarrow^s Q'$ and $P' \approx_i Q'$;

(ii) $Q \Rightarrow^s Q'$ implies that for some P', $P \Rightarrow^s P'$ and $P' \approx_i Q'$.

Then $P \approx Q$ exactly when $\forall i > 0.\, P \approx_i Q$.

We now turn our attention to the behavioral equivalence which the standard notion of set equivalence induces on our model. It turns out that *Proc* equivalence is strictly finer than $\approx_1$ and is incomparable with $\approx_i$ for $i > 1$, meaning that it is incomparable with $\approx$. In what follows $P =_{Proc} Q$ exactly when $T(P) = T(Q)$. The following theorem relies on the well-known observation that $P \approx_1 Q$ exactly when P and Q have the same *observable traces*, where an observable trace of P is an element s of $(A - \{\tau\})^*$ such that there is a P' with $P \Rightarrow^s P'$. Clearly, if P and Q have the same traces then they have the same observable traces.

Theorem 4.1: $=_{Proc}$ is strictly finer than $\approx_1$.

Proof: If $P =_{Proc} Q$ then $traces(P) = traces(Q)$, implying that P and Q have the same observalbe traces and hence $P \approx_1 Q$. Since $T(a(b+c)) \neq_{Proc} T(ab+ac)$ but $a(b+c) \approx_1 ab+ac$, strictness follows.

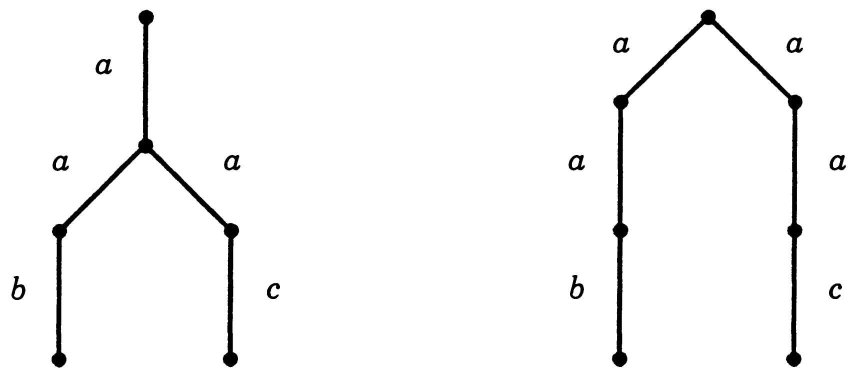
□

The next theorem and corollary demonstrate that $=_{Proc}$ does not fit into the $\approx_i$ hierarchy.

Theorem 4.2: $=_{Proc}$ is incomparable with $\approx_2$.

Proof:

$=_{Proc} \not\subset \approx_2$: Consider the trees P and Q given by $a(ab+ac)$ and $aab+aac$, respectively; see figure 4.2. Clearly $P =_{Proc} Q$ and $\neg(P \approx_2 Q)$.



$a(ab+ac)$ $\qquad$ $aab+aac$

$\{<\varepsilon, \{a\}>, <a, \{a\}>, <aa, \{b\}>, <aa, \{c\}>, <aab, \{\}>, <aac, \{\}>\}$

Figure 4.2. Processes That Are $=_{Proc}$ But Not $\approx_2$.
The *Proc* representation appears below the CCS terms.

$\approx_2 \not\subset =_{Proc}$: Consider the CCS processes P and Q given by τa and a, respectively. $P \approx_2 Q$, and yet since in our model $T(P)$ is $\{<\varepsilon, \{\tau\}>, <\tau, \{a\}>, <\tau a, \{\}>\}$ while $T(Q)$ is $\{<\varepsilon, \{a\}>, <a, \{\}>\}$, $P \neq_{Proc} Q$.

□

In fact the following corollary holds.

Corollary: $=_{Proc}$ is incomparable with observational equivalence.

□

The essential problem is that observational equivalence is not a congruence relation. It does turn out, however, that equality in *Proc* represents a behavioral congruence with respect to CCS, which is to say that if two processes exhibit the same representation in *Proc* then they may be used interchangeably in any context.

Theorem 4.3: $=_{Proc}$ is a congruence.

Proof: If $P =_{Proc} Q$ then it follows from our definitions that for all processes R, $P+R =_{Proc} Q+R$, $P|R =_{Proc} Q|R$, and $P \otimes R =_{Proc} Q \otimes R$ and that for all $a \in A$ $aP =_{Proc} aQ$ and $P\backslash a =_{Proc} Q\backslash a$. Furthermore, if S is a relabeling then $P[S] =_{Proc} Q[S]$.

□

Additionally, $=_{Proc}$ fits into the following strong equivalence hierarchy. Let P and Q be CCS terms, and let $s \in A^*$. Consider the following relations.

$P \simeq_0 Q$.

$P \simeq_{i+1} Q$ for $i \geq 0$ if and only if:

(i) $P \rightarrow^s P'$ implies that for some Q', $Q \rightarrow^s Q'$ and $P' \simeq_i Q'$;

(ii) $Q \rightarrow^s Q'$ implies that for some P', $P \rightarrow^s P'$ and $P' \simeq_i Q'$.

The relation $P \simeq Q$ induced by $\forall i \geq 0.\ P \simeq_i Q$ is the same as strong equivalence, as the following lemmas show.

Lemma 4.1: If $P \sim Q$ and $P \rightarrow^{s} P'$ then there is a Q' such that $Q \rightarrow^{s} Q'$ and $P' \sim Q'$.

Proof: By induction on s.

If s is null then the result is trivial, since in this case $P = P'$; choose Q' to be Q. Now suppose that $s = a\hat{\ }s'$ for $a \in A$ and $s' \in A\ list$. Since $P \rightarrow^{s} P'$ there is a P'' such that $P \rightarrow^{a} P''$ and $P'' \rightarrow^{s'} P'$. From the definition of $\sim$, then, it can be seen that there is a Q'' such that $Q \rightarrow^{a} Q''$ and $Q'' \sim P''$. By the induction hypothesis, there must be a Q' such that $Q'' \rightarrow^{s'} Q'$ and $P' \sim Q'$. Hence the result follows.

□

Lemma 4.2: $P \simeq Q$ if and only if $P \sim Q$.

Proof:

$\Rightarrow$: Assume $P \simeq Q$; we must show that $\forall i \geq 0.\ P \sim Q$. The proof will proceed by showing that if $P \simeq_i Q$ then $P \sim_i Q$; given this, the result follows since $\forall i \geq 0.\ P \simeq_i Q$. The proof will proceed by induction on i. Clearly, if $P \simeq_0 Q$ then $P \sim_0 Q$. Now fix $i \geq 0$ and assume that $P \simeq_i Q$ implies that $P \sim_i Q$; it remains to show that if $P \simeq_{i+1} Q$ then $P \sim_{i+1} Q$. Assume $P \simeq_{i+1} Q$. If $P \rightarrow^{a} P'$ then since $P \simeq_{i+1} Q$ there must be a Q' such that $Q \rightarrow^{a} Q'$ and $P' \simeq_i Q'$. By induction $P \sim_i Q$, and as the symmetrical result holds we may therefore conclude that $P \sim_{i+1} Q$.

$\Leftarrow$: Assume $P \sim Q$; we must show that $\forall i \geq 0.\ P \simeq_i Q$. The result follows from induction on i. Clearly, $P \simeq_0 Q$. Now fix $i \geq 0$ and assume that $P \simeq_i Q$; we must show that $P \simeq_{i+1} Q$. To do so, assume $P \rightarrow^s P'$; we show that there exists Q' with $Q \rightarrow^s Q'$ and $P' \simeq_i Q'$. By lemma 4.1, there exists a Q' such that $Q \rightarrow^s Q'$ and $P' \sim Q'$, and by the induction hypothesis $P \simeq_i Q$. As the symmetrical result holds, we may conclude that $P \simeq_{i+1} Q$.

$\square$

The following theorem places $=_{Proc}$ in the $\simeq_i$ hierarchy.

Theorem 4.4: $\simeq_1$ is strictly coarser than $=_{Proc}$, which is strictly coarser than $\simeq_2$.

Proof:

$=_{Proc} \subset \simeq_1$: $\simeq_1$ is just trace equality, so if $P =_{Proc} Q$ then $P \simeq_1 Q$. Strictness follows from the fact that $a(b+c) \simeq_1 (ab+ac)$ but $a(b+c) \neq_{Proc} (ab+ac)$.

$\simeq_2 \subset =_{Proc}$: Suppose $P \simeq_2 Q$; we must show that $P =_{Proc} Q$. It is sufficient to show that $T(P) \subseteq T(Q)$ and that $T(Q) \subseteq T(P)$. To this end, assume that $<s, N> \in T(P)$; it is necessary to show that $<s, N> \in T(Q)$. Since N is a next set associated with s, there is a P' such that $P \rightarrow^s P'$ and such that there exists a P'' with $P' \rightarrow^a P''$ for $a \in A$ if and only if $a \in N$. Moreover, since $P \simeq_2 Q$, for all P' such that $P \rightarrow^s P'$, there is a Q' such that $Q \rightarrow^s Q'$ and $P' \simeq_1 Q'$. Therefore, there must exist a Q' such that $Q \rightarrow^s Q'$ and $Q' \simeq_1 P'$.

This last fact implies that for $a \in A$, there is a Q'' such that $Q' \rightarrow^{a} Q''$ exactly when $a \in N$. This in turn establishes that $<s, N> \in T(Q)$. A symmetric argument establishes that $Q \subseteq P$, and we therefore have that $P =_{Proc} Q$. Strictness follows from the fact that $a(ab+ac) =_{Proc} (aab+aac)$ but $\neg[a(ab+ac) \simeq_2 (aab+aac)]$.

□

This theorem implies the following corollary.

Corollary: $=_{Proc}$ is strictly coarser than $\sim$.

□

4.3.2. A Comparison with Other Equivalences

This section compares equality in *Proc* with two other equivalences, *failures* equivalence and *testing* equivalence. The relation $=_{Proc}$ proves to be finer than each of these equivalences, and this fact implies that objects in the *Proc* model have a natural translation into the failures model and into the testing model. In what follows let $F(P)$ be the failures model of process P, and let $T_e(P)$ be the testing model of process P. $=_F$ will denote equivalence in the failures model, while $=_T$ denotes equivalence in the testing model.

We first consider failures equivalence, which will be denoted $=_F$.

Theorem 4.5: $=_{Proc}$ is finer than $=_F$.

Proof: Suppose $P =_{Proc} Q$; we must show that $P =_F Q$. Since $P =_{Proc} Q$ it follows that $traces(P) = traces(Q)$; therefore, proving that $P =_F Q$ entails proving that if $s \in traces(P)$ then the refusals associated with s in $F(P)$ and $F(Q)$ are the same. For a process R and a trace s, define $F_R(s) = \cup\{X \mid <s,X> \in F(P)\} - \{a \mid \neg(s\hat{\ }a \in traces(P))\}$. Intuitively, $F_R(s)$ consists of exactly those actions that R may either refuse or accept after executing trace s. Because of the closure properties associated with refusals, it suffices to show that the set $F_P(s) = F_Q(s)$. Let $N_1, ..., N_k$ be the next sets associated with s in $T(P)$ and $T(Q)$. Then $F_P(s) = \cup((\cup N_i) - N_i) = F_Q(s)$, and $P =_F Q$. That is, the equivalence in the acceptance model is at least as fine as failures equivalence. Strictness follows from the observation that $aa + a(a+b) + a(a+b+c)$ and $aa + a(a+b) + a(a+c)$ are failures-equivalent but not *Proc*-equivalent, where we are assuming that $A = \{a,b,c\}$.

□

Testing equivalence, which will be denoted by $=_T$, states that two processes are equivalent exactly when, at any point in a computation, the actions they *must* accept are the same and the actions they *may* accept are the same. This equivalence is also strictly coarser than $=_{Proc}$.

Theorem 4.6: $=_{Proc}$ is finer than $=_T$.

Proof: Suppose $P =_{Proc} Q$; we must show that $P =_T Q$. Since $P =_{Proc} Q$ it follows that $traces(P) = traces(Q)$; therefore, for a given trace s, it suffices to show that $must_P(s) = must_Q(s)$ and $may_P(s) = may_Q(s)$, where $must_R(s)$ is the set of actions R must accept after traces s and $may_R(s)$ is the set of actions R may accept after s. Let $N_1, \ldots, N_k$ be the next sets associated with s in $T(P)$ ($T(Q)$). Then $must_P(s) = \cap N_i = must_Q(s)$ and $may_P(s) = \cup N_i = may_Q(s)$. Therefore, $P =_T Q$. Strictness follows from the same example as before; $aa + a(a+b) + a(a+b+c)$ and $aa + a(a+b) + a(a+c)$ are testing equivalent but not *Proc*-equivalent.

□

So far this section has demonstrated that although $=_{Proc}$ is not as strong as strong equivalence in CCS, *Proc* is an interesting model in that other models of concurrency--the failures model and the testing model--have equivalences that are implied by equivalence in *Proc*. We conclude the discussion with some remarks about $=_{Proc}$ in comparison with $\sim$. It has been argued [Strom 86] that strong equivalence is too strong for practical purposes. Consider the CCS processes $\alpha(\beta\gamma NIL + \beta\delta NIL)$ and $\alpha\beta\gamma NIL + \alpha\beta\delta NIL$. These two processes are not strongly equivalent. However, one reasonable model of computation specifies that a machine at any point in its computation makes known to the user the actions that are enabled next. Under this model of computation the two processes cannot be distinguished. Each can accept an α and then a β, after which either a γ or a δ, but not both, are possible. In fact, these processes are equal in the acceptance model, as figure 4.3 shows. It can be argued that $=_{Proc}$ fully characterizes the

$$\alpha(\beta\gamma NIL + \beta\delta NIL) =$$
$$\{ <\varepsilon, \{\alpha\}>, \quad <\alpha, \{\beta\}>, \quad <\alpha\beta, \{\gamma\}>, \quad <\alpha\beta, \{\delta\}>, \quad <\alpha\beta\gamma, \{\}>, \quad <\alpha\beta\delta, \{\}>\}$$
$$\alpha\beta\gamma NIL + \alpha\beta\delta NIL =$$
$$\{ <\varepsilon, \{\alpha\}>, \quad <\alpha, \{\beta\}>, \quad <\alpha\beta, \{\gamma\}>, \quad <\alpha\beta, \{\delta\}>, \quad <\alpha\beta\gamma, \{\}>, \quad <\alpha\beta\delta, \{\}>\}$$

Figure 4.3. *Proc* Equivalence of Two CCS Expressions.

computations possible on this "one-action lookahead" machine.

4.4. Modeling Temporal Logic

Chapter 2 indicated the interest that temporal logic has received as a language for stating and proving properties of concurrent programs. This section shows how temporal logic can be used to reason about elements in *Proc*. To do so the section describes the semantics of temporal fomulas in terms of objects in *Proc* and then notes how properties may be specified using these semantics.

4.4.1. Syntax and Semantics

The syntax of temporal logic used in this section is similar the one described in chapter 2. Let Φ be the set of atomic formulas, with $\phi \in \Phi$. The temporal formulas have the following description.

$$
\begin{array}{lll}
\Gamma & ::= & \phi \\
 & | & \neg\Gamma \\
 & | & \Gamma \wedge \Gamma \\
 & | & \Box\Gamma \\
 & | & \bigcirc\Gamma
\end{array}
$$

The propositional connectives $\wedge$ and $\neg$ have the obvious meaning. $\Box\,\Gamma$ means "henceforth Γ"; proposition Γ is meant to hold throughout the rest of the programs execution. $\bigcirc\Gamma$ means "at the next instant, Γ"; at the next instant in the process's execution, the proposition Γ holds. Other connectives and operators, including $\vee$, $\Rightarrow$ and $<>$, the eventuality operator, can be defined in terms of the operators presented above.

The following definition will simplify the presentation of the formal semantics of the logic. Given P an element of *Proc* and a trace s, $P \backslash s$ refers to the element of *Proc* that P may evolve to after executing s.

$$P \backslash s = \{<t, N> \mid <s\,\hat{}\,t, N> \in P\}$$

If $s = \varepsilon$ then $P \backslash s = P$.

The formal semantics of these temporal formals have the following specification in the *Proc* model. The semantic relation $\models$ denotes satisfiability; $P \models \Gamma$ expresses the notion that formula Γ holds for process P.

Let the elements of Φ be interpreted as predicates on *finset(A)*; that is, if $\phi \in \Phi$ then ϕ*:finset(A)*$\rightarrow$*{True, False}*. Intuitively, ϕ will apply to the next sets of an element of *Proc*.

$P \models \phi$ if and only if $\forall s \in \{x: finset(A) \mid <\varepsilon, x> \in P\}.\ \phi(s)$

A process satisfies a primitive formula if its initial next sets satisfy ϕ.

$P \models \neg \Gamma$ if and only if it is not the case that $P \models \Gamma$

A process satisfies $\neg \Gamma$ when it is does not satisfy Γ.

$P \models \Gamma_1 \wedge \Gamma_2$ if and only if $P \models \Gamma_1$ and $P \models \Gamma_2$

A process satisfies a conjunction of formulas when it satisfies each conjunct.

$P \models \Box \Gamma$ if and only if $\forall s \in traces(P).\ P \backslash s \models \Gamma$

A process satisfies "henceforth Γ" when it satisfies Γ and any process that it can evolve into during the course of its execution satisfies Γ.

$P \models \bigcirc\Gamma$ if and only if $\forall a \in \{x \in A \mid \exists M.\ <x, M> \in P\}.\ P \backslash a \models \Gamma$

A process satisfies "at the next instant, Γ" when each process that it may evolve to after one atomic action satisfies Γ.

4.4.2. Examples of Temporal Specification in *Proc*

This section considers some examples of both safety and liveness properties [Lamport 77] that can be specified in the semantic description of temporal logic in *Proc* given previously. Our first example involves stopping. A process is *stopped* if it can perform no actions; it may either be terminated or deadlocked. To introduce the notion of *stopped* into the temporal logic defined above, define *stopped* as a primitive predicate in the following fashion.

$$stopped \;=\; \lambda s.s=\varnothing$$

That is, a process is stopped at a point in its execution if the next set corresponding to that point of the execution is empty. The following temporal formula is only satisfied by processes that always stop.

$$\Box <>stopped$$

The next example describes necessity of action; this notion can be used to express notions of fairness. For an action a, define *must(a)*, a primitive forumula, as follows.

$$must(a) \;=\; \lambda s.s=\{a\}$$

Intuitively, a process must perform an a action at a point in its execution if the next set at that point is the singleton set $\{a\}$. The following expresses the property that it is always the case that a process must eventually perform an a action.

$$\Box <>must(a)$$

Given a choice operator, **or**, we may now say that it is *fair* if the program

repeat *(a* **or** *b)*

satisfies the formula $(\Box <>must(a)) \wedge (\Box <>must(b))$.

The final property is a safety property; it is noteworthy in that it shows how one can reason about possibility using the logic given above. Consider the following primitive formula.

$$may(a) \;=\lambda s.a\in s$$

This predicate holds of a process when it may perform an a action. The following formula holds when an a action is always available to a process.

$\square\, may(a)$

We close this section by observing that it is impossible to model a choice operator **choose** that is *weakly* fair in the sense that the program

$P \equiv$ **repeat choose**(a,b)

is capable of performing all fair sequences of a and b actions. Suppose defining **choose** were possible; then P would satisfy the following formula.

$(\square\!<\,>must(a)) \wedge (\square\!<\,>must(b)) \wedge (\square\, may(a)) \wedge (\square\, may(b))$

No element of *Proc* can satisfy this, however, because it is impossible to have a next set that satisfies both $must(a)$ and $may(b)$.

4.5. Expressing *Proc* as a Type

So far this chapter has examined properties of *Proc* without describing how *Proc* fits into the main theme of the thesis, the type-theoretic analysis of concurrency. This section addresses this by describing how Nuprl may be used as a tool for reasoning about objects in *Proc*. The intuitive approach is to develop a type, *Proc*, whose elements are the elements of *Proc* and whose equality relation is $=_{Proc}$; using the Nuprl deductive apparatus, then, one can develop mechanisms that enable properties, including temporal properties, to be expressed and proven of elements of *Proc*. This section is not intended to be exhaustive, but rather suggestive.

We shall begin the exposition with a development of some rudimentary set theory in Nuprl.

4.5.1. Sets in Nuprl

Since Nuprl does not have a primitive notion of set, and since the objects of *Proc* will be sets, a rudimentary modeling of set theory is necessary. The development proceeds in the following fashion. Intuitively, sets over a base type T will be identified with their characteristic functions, which will be type-valued functions with T as their domain and with the interpretation that an element of T is a member of a set exactly when the type resulting from the application of the characteristic function is *inhabited*. Since equality between functions does not correspond to the notion of set equality, the equality between these functions will need to be redefined using the quotient type constructor described in chapter 3; briefly, this type constructor allows the redefinition of equality over a base type. Given a type T and an equality relation $E(x,y)$ over x and y, the type $(x,y){:}T//E(x,y)$ consists of the elements of T with equality between elements declared to be E. Accordingly, define *Set* to be a constructor that takes a base type and returns a type of sets of T.

$$Set{:}\ \ U_1 \rightarrow U_2$$

$$= \lambda T.(P,Q){:}T \rightarrow U_1 \,//\, \forall x{:}T.P(x) \Leftrightarrow Q(x)$$

For a type T, $Set(T)$ returns a quotient type whose elements are functions from T to the types in U_1 and whose equality corresponds to extensional set equality.

Given $Set(T)$, various pieces of set theory can be defined in the following fashion. In what follows S_i will denote a set of elements of type T, while $\underline{S}_i$ will denote the representation of S_i in $Set(T)$. For a of type T, $a \in S_1$ corresponds to $\underline{S}(a)$. $\underline{\varnothing}$ is $\lambda x.void$, and for an element a of T $\underline{\{a\}}$ is $\lambda x.(x=a \; in \; T)$. $\underline{S_1 \cup S_2}$ is $\lambda x.(\underline{S}_1(x)|\underline{S}_2(x))$ (since $\underline{S}_1(x)|\underline{S}_2(x)$ is inhabited exactly when either $\underline{S}_1(x)$ or $\underline{S}_2(x)$ is), and $\underline{\cup S_i}$ where i is of type I is $\lambda x.\exists i{:}I.\underline{S}_i(x)$. Given a predicate $P{:}T \rightarrow U_1$, $\underline{\{x \in S_1 | P(x)\}}$ is $\lambda x.(\underline{S}_1(x)\#P(x))$, since $\underline{S}_1(x)\#P(x)$ is inhabited exactly when both $\underline{S}_1(x)$ and $P(x)$ are. For a function $f{:}T \rightarrow U$, $\underline{f(S_1)}$ is $\lambda x.(\exists a{:}T.\underline{S}_1(a) \;\&\; f(a)=x \; in \; U)$.

4.5.2. Defining *Proc* using *Set*

The next step in defining *Proc* as a type involves defining a type corresponding to trace-next set pairs. Given a type A of actions, the type A *list* has as its elements the elements of A^*, and so traces will come from A *list*. The empty traces, ε, is modeled as *nil*. Concatenation (^) can be defined in the following fashion.

$$concat{:} \quad A\ list \rightarrow A list \rightarrow A list$$
$$= \lambda l1.\lambda l2.list\_ind(l1;l2;h,t,i.(h.i))$$

Chapter 6 contains an account of a constructor *finset*, which for a U_1-type T corresponds to a type of finite sets of T. Next sets will be modeled as elements of *finset(A)*. The type of trace-next set pairs may now be defined as follows.

$$comps: \quad U_1$$
$$= A \; list \; \# \; finset(A)$$

Since elements in *Proc* are sets of *comps* satisfying properties (1)-(3) described earlier in this chapter, the following Nuprl subtype describes the *Proc*.

$$Proc = \{P:Set(comps) \mid$$
$$\exists s:finset(Action).P(<nil, s>) \;\&$$
$$\forall x:comps.\; P(x) \Rightarrow \forall a \in x.2.\exists s:finset(Action).P(<x.1\,\hat{}\,a, s>) \;\&$$
$$\forall x:comps.\; P(x) \& x \neq nil \;\Rightarrow\; \exists y:comps.\; P(y) \;\&\; \exists a \in y.2.\; y.1\,\hat{}\,a$$
$$= x \; in \; A \; list\}$$

The three conjuncts correspond to conditions (1)-(3), respectively.

Several of the functions used in the chapter may be defined formally in the Nuprl environment. For example, *traces* may be defined as a type-valued function that, for a given element *P* of *Proc*, returns a type whose elements are the traces of *P*.

$$traces: \quad Proc \rightarrow U_1$$
$$= \lambda P.\; \{l:A \; list \mid \exists s:finset(A).\; P(<l,s>)\}$$

The next sets associated with a trace in a process may be defined by the following function.

$$next \;:\; Proc \rightarrow A \; list \rightarrow U_1$$
$$= \lambda P.\lambda l.\{s:finset(A) \mid P(<l,s>)\}$$

It is also possible to present the encoding of the semantics for temporal logic in the Nuprl account of *Proc* and to develop a series of tactics that, using the the primitive inference rules that the Nuprl system provides, implement the inference rules of temporal logic given in chapter 2. Using these tactics, it would then be possible to reason in a machine-assisted fashion about the temporal properties of elements of *Proc*.

Chapter 5

A Recursive Set Model

The *Proc* model has many pleasing properties. It has a simple set-theoretic explanation, and the translation of the set-theoretic explanation into a type-theoretic account in the Nuprl environment is relatively straightforward and easy to implement. Furthermore, this model has enough expressive power to serve as a model for various modal logics used to reason about concurrency; in particular, one can give a semantic account of temporal logic and then use the proof system associated with the logic to reason about processes expressed as objects of *Proc*. Using *Proc* as a semantic basis, developing a theory of processes and logics for reasoning about them is straightforward in Nuprl; the result of doing so would be an automated assistant for reasoning about processes.

However, the *Proc* model also has drawbacks, and addressing these drawbacks leads to the model presented in this chapter. One problem with the *Proc* model is the coarseness of its equivalence; the fact that the equivalence is weaker than strong equivalence and incomparable with observational equivalence makes it unsuitable as a model for CCS. Another problem with the *Proc* model stems from the fact that its representation of processes obscures the simple inductive definition that most process constructors have. The somewhat complicated definition of CCS composition, especially in light of its simple semantic description, highlights the fact that the *Proc* model does not provide a natural means for expressing inductive definitions on processes.

A model with recursively defined objects and a stronger equivalence than *Proc* equivalence is therefore desirable. In fact, we argue that a model whose equality corresponds to strong equivalence is a very natural and desirable one. A structure with this equivalence relation between elements will model both the strong and observational equivalences, as well as the observational congruence, of CCS, since observational equivalence and observational congruence are strictly weaker than strong equivalence. Moreover, strong equivalence has an elegant mathematical characterization that simplifies metareasoning about it.

The chapter is organized along the following lines. The first section presents a set-theoretic formalization of the new model, and the second section contains a formalization of CCS in the model. The section following consists of a proof that equivalence in the model corresponds to the CCS notion of strong equivalence. The remaining sections comprise examples designed to illustrate the different styles of reasoning accommodated by this model, including the algebraic style proposed by advocates of CCS and more logic-based styles favored by other researchers. Mention is also made of the use of Nuprl as a tool for developing an environment for reasoning about objects in the model. The final section outlines how a particular modal logic can be developed in Nuprl.

5.1. A Presentation of the Model

The model to be presented was first developed by the author in a type-theoretic framework in 1985 [Cleaveland and Panangaden 85] and is very similar to a set-theoretic model developed independently by Aczel [Aczel 85]. Intuitively, processes are represented as finite sets of pairs, where each pair consists of an action and a continuation, a continuation being itself a process. At any point in its execution, a process may have several choices of action available to it, and the result of performing an action determines what the remainder of the computation of the process will be.

The possibility of nonterminating computations complicates the set-theoretic development of this model somewhat, as Aczel notes. Standard Zermelo-Fraenkel set theory proscribes the existence of so-called "non-well-founded" sets, which are sets having infinite $\in$-chains. However, nontermintating processes in the model just described give rise to situations of the form $<a,P>\in Q \wedge <b,Q>\in R \wedge \ldots \wedge <c,S>\in P$, which is an example of an infinite $\in$-chain. The regularity (often called the foundation) axiom is responsible for the prohibition of these sets, so by modifying ZFC by replacing this axiom with a weaker, "anti-foundation" axiom due to Aczel, non-well-founded sets may be allowed. As section 5.1.1 shows, the resulting set theory is consistent.

Aczel's process model is more general than ours in that he does not enforce finite choice at any point in the model; however, the computational subtlety of implementing infinite sets in an appropriate way in Nuprl, as well as our belief that it is natural to enforce finite choice during a process's execution, makes us opt for finite choice.

5.1.1. Non-Well-Founded Set Theory

This section briefly describes pertinent aspects of the non-well-founded set theory due to Aczel [Aczel 85]. It presents the anti-foundation axiom (AFA) in detail and describes the effect that introducing the axiom has on equality between sets.

Sets can be identified with cetain kinds of directed graphs called *accessible pointed graphs*, or apgs short. Apgs are characterized by a distinguished node, the *point*, and by the property that there is a path from the point to every node in the graph. Intuitively, the point denotes the set being defined; nodes represent members, and edges denote membership, so that $a \rightarrow b$ exactly when $b \in a$. Figure 5.1 presents apgs corresponding to the ordinal representations of the natural numbers *0, 1, 2* and *3*. Note that *2* and *3* each have two different apg representations; in general, a set may be described by many different apgs.

A *decoration* of an apg is is an assignment of a set to each node in the graph in such a way that the elements of a set assigned to a node are the sets assigned to the children of the node. A *picture* of a set is an apg with a decoration having the set assigned to the point. Figure 5.2 shows the

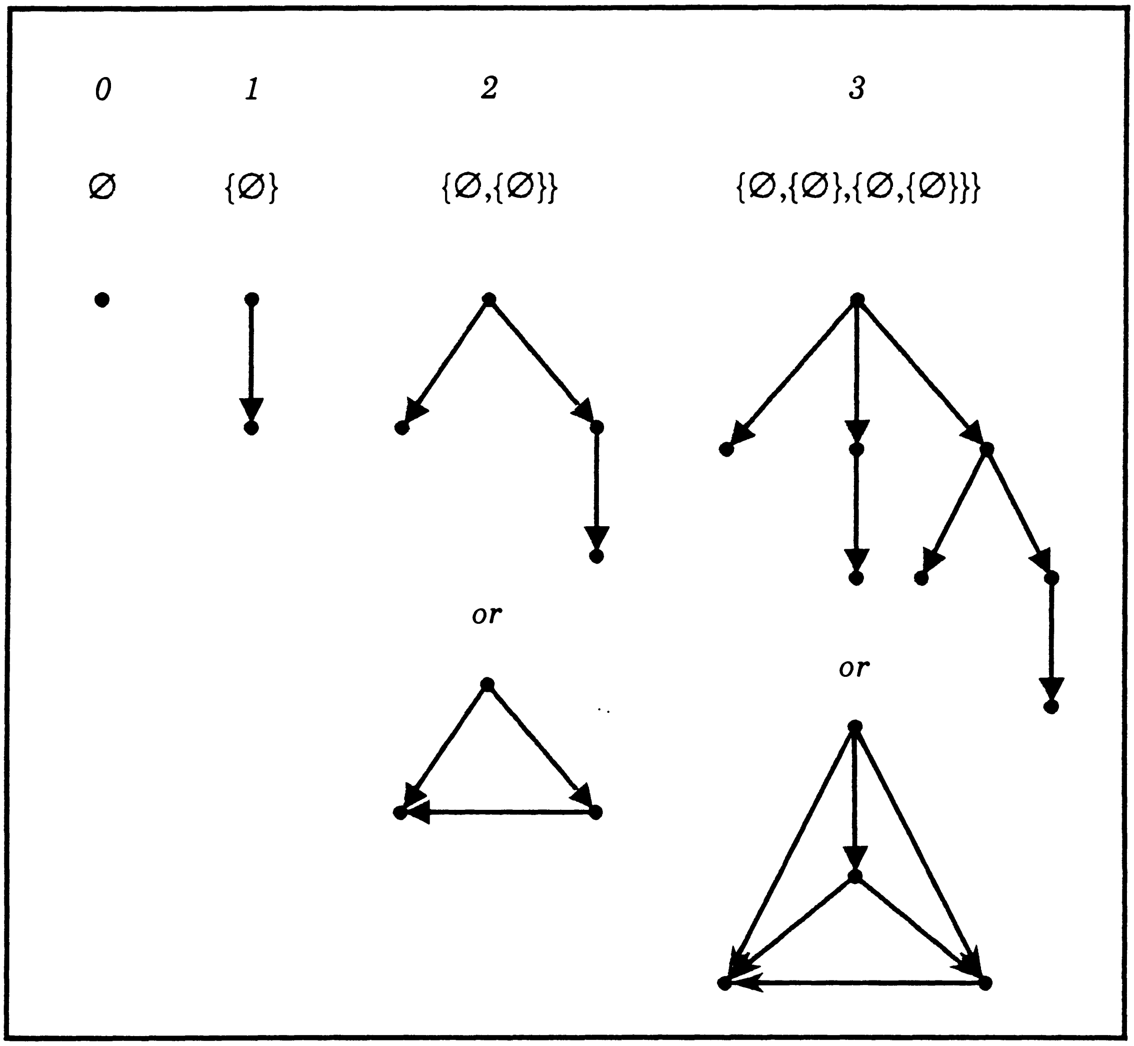
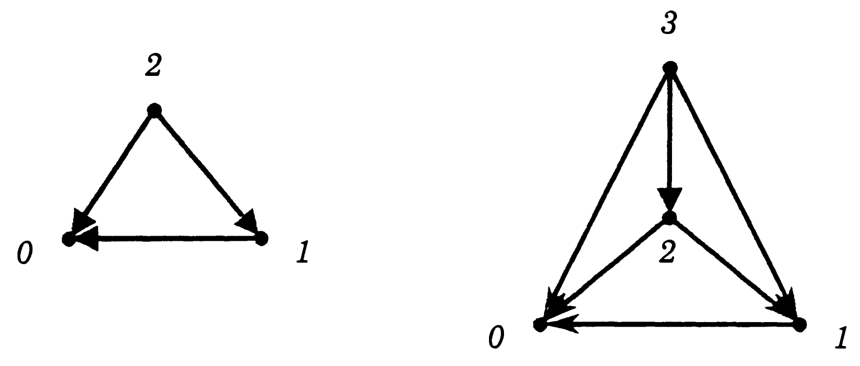


Figure 5.1. Accessible Pointed Graphs Corresponding to *0*, *1*, *2* and *3*. The ordinal representations of these numbers are also given.

pictures of the ordinals corresponding to *2* and *3*.

Several observations are in order. The empty set corresponds to the apg having no edges. Also, in the examples we have seen, each apg has a unique decoration. Define an apg to be *well-founded* if and only if it has no infinite path.



The picture of *2*. The picture of *3*.

Figure 5.2. Examples of Pictures.

1. Every well-founded graph has a unique decoration.

   (This is the Mostowski Collapsing Lemma [Aczel 85].)

2. Every well-founded apg is a picture of a unique set.
3. Every set has a picture.

These results are shown by Aczel. The statement of the anti-foundation axiom is the following.

Anti-foundation Axiom: Every apg has a unique decoration.

Thus every apg is the picture of a unique set, and non-well-founded sets exist; in fact, any non-well-founded apg is the picture of a non-well-founded set. Figure 5.3 shows examples of non-well-founded sets and their pictures. The first picture corresponds to the set *{{{{{...}}}}}*, while the second

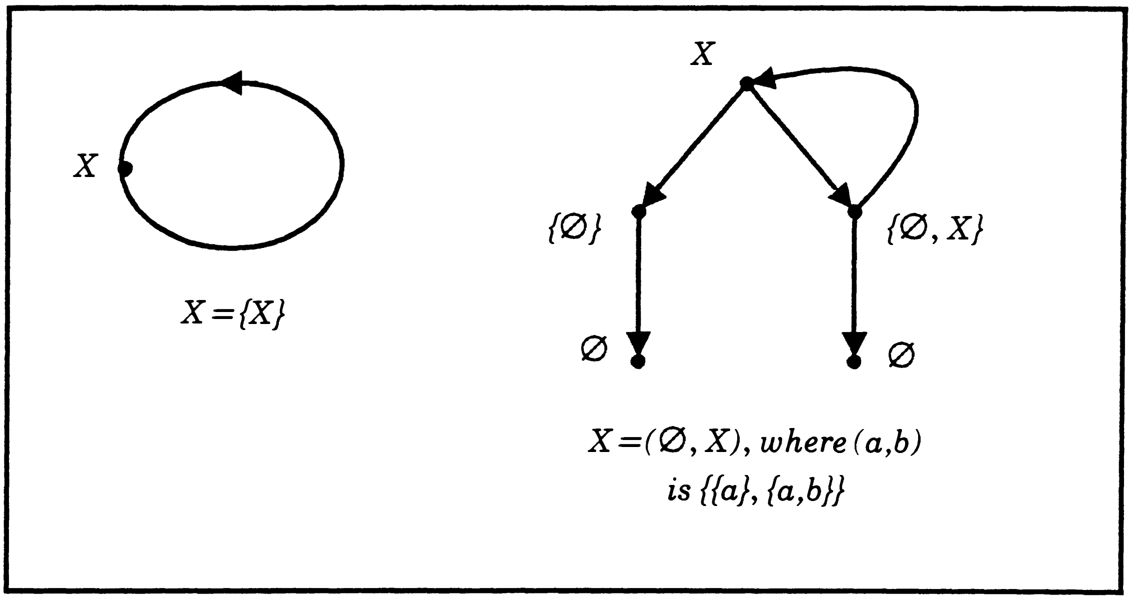


Figure 5.3. Non-Well-Founded Sets and Their Pictures.

corresponds to the "infinite stream" of $\emptyset$'s, $(\emptyset, (\emptyset, ...))$. The equational specifications given in the figure are interesting in that the anti-foundation axiom specifies a unique solution to them.

The set theory resulting from the axioms of extensionality, pairing, union, powerset, infinity, separation, collection, choice and AFA is consistent, with apgs serving as the model. Let ZFC$-$ + AFA denote this theory; for convenience, the axioms are listed in figure 5.4.

The anti-foundation axiom also has consequences for set equality. Aczel shows that the axiom implies that set equality is *strongly extensional*, where strong extensionality is defined in the following fashion. We will first define the notion of bisimulation. Let s and t be sets; then a relation R is a *bisimulation* when sRt implies the following.

(i) $\forall x' \in x.\ \exists y' \in y.\ x'Ry'$.

Extensionality	$\forall z\,(z\in a \Leftrightarrow z\in b) \Rightarrow a=b$
Pairing	$\exists z\,(a\in z\ \&\ b\in z)$
Union	$\exists z \forall x\in a \forall y\in x\,(y\in z)$
Powerset	$\exists z \forall x\,(\forall u\in x\,(u\in a) \Rightarrow x\in z)$
Infinity	$\exists z\,(\exists x\in z\ \forall y\,(y\notin x)\ \&\ \forall x\in z\ \exists y\in z\,(x\in y))$
Separation	$\exists z \forall x\,(x\in z \Leftrightarrow z\in a\ \&\ \phi)$
Collection	$\forall x\in a\ \exists y\ \phi \Rightarrow \exists z \forall x\in a\ \exists y\in z\ \phi$
Choice	$\forall x\in a\ \exists y\,(y\in x)\ \&\ (\forall x_1,x_2\in a)(\exists y(y\in x_1\ \&\ y\in x_2) \Rightarrow x_1=x_2) \Rightarrow \exists z \forall x\in a\ \exists! y\in x\,(y\in z)$
Anti-foundation	*Every apg has a unique decoration.*

Figure 5.4. The Axioms of ZFC− + AFA.
ϕ is any formula in which z does not occur free.

(ii) $\forall y'\in y.\exists\, x'\in x.\ x'Ry'$.

It turns out that there is a unique weakest bisimulation $\equiv$ that is the union of all bisimulations. Equality is *strongly extensional* if it is the case that $s\equiv t$ implies that $s=t$. Aczel shows the following (which we paraphrase).

AFA is valid if and only if equality is strongly extensional.

5.1.2. The Model of Processes

Using the brief description of non-well-founded sets given in the previous section we now describe the process model. Let *finset(X)* denote the collection of finite sets of set X. The set ST, defined as the unique solution guaranteed by ZFC− +AFA to following equation, represents the set of processes.

$$ST = finset(A \times ST)$$

That is, elements of ST are finite sets of pairs, where the first element is a member of A and the second element is an element of ST.

The members of ST are intended to be interpreted as (potentially) nondeterministic processes, with the set A representing the set of actions a process may perform. The element of ST corresponding to a process describes the courses of computation available to the process as pairs, where the first element of each pair represents an action the process may perform and the second element of each pair represents the rest of the computation available to the process after engaging in the associated action. Under this interpretation, non-well-founded elements of ST correspond to processes that may not terminate, while the well-founded elements of ST describe processes that must terminate. Nondeterminism can result from the fact that the same action can occur in two pairs having distinct second elements. Figure 5.5 describes examples of CCS expressions represented as in ST.

Some remarks about developing ST in Nuprl are in order here. Assuming the existence of a type constructor *finset*, where elements of *finset(T)* for a given type T are the finite sets of elements of type T, one may use the recursive type constructor supplied by the Nuprl theory to build a type that is a subset of ST. This type, *rec(st.finset(Action#st))*, is the least solution (with respect to a containment ordering) to the type equation $st = finset(Action \times st)$, and as such it only contains well-founded sets. A

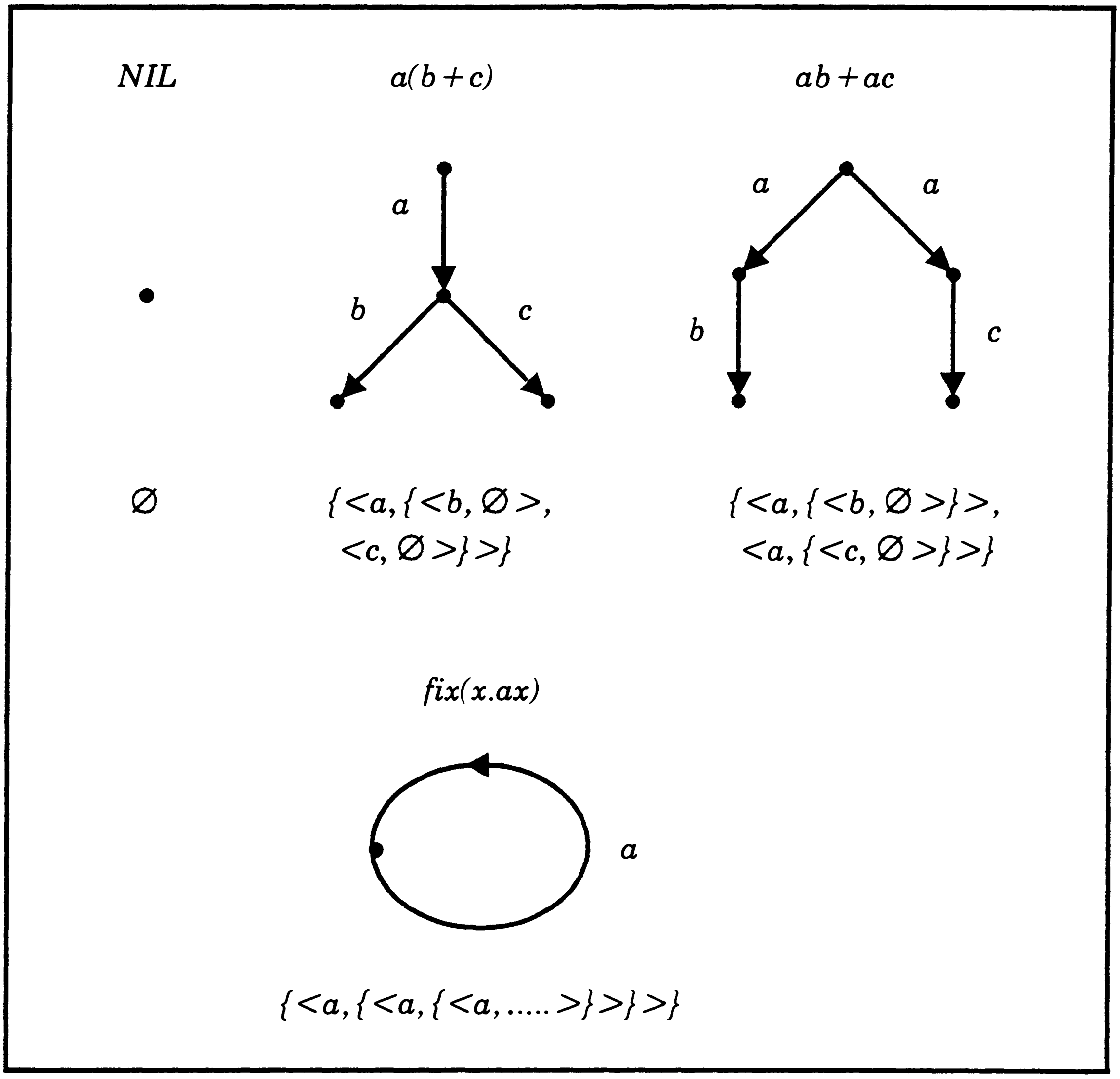


Figure 5.5. *ST* Representations of CCS Terms.
The picture appears below the term; the representation appears below the picture.

proposed infinite type constructor [Mendler, Panangaden and Constable 86] would allow nonterminating processes to be represented. The type *inf(st.finset(Action#st))* represents the maximal solution to above type equation and therefore includes the non-well-founded sets as well as the well-founded ones. Since *inf* types have not been implemented the implementation in Nuprl described in chapter 6 uses *rec* types.

5.2. A Development of CCS Operators in the Model

This section gives accounts of the CCS term constructors in ST. In order to reason for the correctness of the developments of CCS constructors we must also give an account of the CCS semantic transition relation $\rightarrow^a$. This can be accomplished by modeling $P \rightarrow^a P'$ as $<a, S(P')> \in S(P)$, where $S(P)$ represents the ST representation of P. Using the set-theoretic constructions of ZFC− +AFA, we can now describe CCS processes P in terms of their representations in ST and argue for correctness of the representations as follows.

Nil: $S(NIL) = \varnothing$. Since $\varnothing$ contains no elements this correctly describes the fact that NIL performs no actions.

Prepend: $S(aP) = \{<a, S(P)>\}$. Since the only member of this set is $<a, S(P)>$, it follows that $aP \rightarrow^a P$.

Choice: $S(P+Q) = S(P) \cup S(Q)$. That this is correct stems from the definition of union. Suppose $P \rightarrow^a P'$, meaning that $P+Q \rightarrow^a P'$. Then $<a, S(P')> \in S(P)$ and $<a, P'> \in S(P) \cup S(Q)$. Similarly, if $Q \rightarrow^a Q'$, meaning that $P+Q \rightarrow^a Q'$, then $<a, S(Q')> \in S(Q)$ and $<a, S(Q')> \in S(P) \cup S(Q)$.

Restriction: Define an operation *prune* on elements a of A and elements s of ST using set comprehension in the following fashion.

$$prune(a, s) = \{<b, prune(a, t)> \mid <b, t> \in s \;\&\; b \neq a\}$$

Since for any a and s this equation can be represented as an apg AFA guarantees a unique solution. Now,

$$S(P\backslash a) = prune(a, prune(\underline{a}, S(P))).$$

This correctly describes CCS restriction, for if $P \rightarrow^{b} P'$ and b is not a or $\underline{a}$, implying that $P'\backslash a \rightarrow^{b} P'\backslash a$, then $<b, S(P')> \in S(P)$ and $<b, prune(\underline{a}, prune(a, S(P')))> = <b, S(P'\backslash a)> \in S(P\backslash a)$.

Relabeling: Define an operation *apply* on functions f from A to B and elements s of ST as follows.

$$apply(f, s) = \{<f(a), apply(f, t)> \mid <a, t> \in s\}$$

This operation is valid in ZFC− +AFA; it is just an instance of collection. Now, if L is a relabeling from action set A to action set B (i.e. $L(\tau) = \tau$ and $L(\underline{a}) = \underline{L(a)}$) then $P[L]$ may be modeled in the following fashion.

$$S(P[L]) = apply(L, S(P))$$

Clearly, if $P \rightarrow^{a} P'$, implying that $P[L] \rightarrow^{L(a)} P'[L]$, then $<a, S(P')> \in S(P)$ and $<L(a), S(P'[L])> \in S(P[L])$.

Composition: Define *comp* as an operation on two elements of ST in the following way.

$$\begin{aligned} comp(s, t) = \; & \{<a, comp(s', t)> \mid <a, s'> \in s\} \quad \cup \\ & \{<b, comp(s, t')> \mid <b, t'> \in t\} \quad \cup \\ & \{<\tau, comp(s', t')> \mid \exists a.\; <a, s'> \in s \;\&\; <\underline{a}, t'> \in t\} \end{aligned}$$

This is a valid operation on ST, involving instances of collection and comprehension. Now it is clear that the following is correct.

$$S(P|Q) = comp\,(S(P), S(Q)).$$

Recursion: For x a free variable, define $S(x) = x$, and define substitution in the obvious way. Now,

$$S(fix(x.b)) = S(b)[S(fix(x.b))/x].$$

Since b must be guarded this expression is well-formed in the theory of non-well-founded sets. It is clear that it satisfies the semantic specification for recursive definition.

We may also define SCCS composition and the CSP composition operators. To define SCCS composition let A be an Abelian group with $\circ$ the group operation and τ the group identity, and let

$$scomp\,(s, t) = \{<a\circ b, scomp\,(s', t')> \mid <a, s'> \in s \;\&\; <b, t'> \in t\}.$$

Then

$$S(P \otimes Q) = \quad scomp\,(S(P), S(Q)).$$

The CSP operations may be defined in a similar fashion, except for $+$, which has no obvious representation in ST.

5.3. Analyzing Equality in the Model

This section proves that equality in ST is exactly strong equivalence. Let $P =_{ST} Q$ exactly when $S(P) = S(Q)$, where equality on non-well-founded sets is the same extensional equality found on well-founded sets. The proof uses the fact that $\sim$ may be characterized as the weakest *strong bisimulation* between processes [Milner 83]. A relation R is a *strong bisimulation* when for processes P and Q, the following hold when $P\ R\ Q$.

(i) If $P \rightarrow^a P'$ then there exists Q' such that $Q \rightarrow^a Q'$ and $P'\ R\ Q'$.

(ii) If $Q \rightarrow^a Q'$ then there exists P' such that $P \rightarrow^a P'$ and $P'\ R\ Q'$.

Milner [Milner 83] shows that $P \sim Q$ exactly when there exists a strong bisimulation R such that $P\ R\ Q$; in other words, $\sim$ is the union of all strong bisimulations.

Theorem 5.1: $P =_{ST} Q$ if and only if $P \sim Q$.

Proof: The proof is in two parts.

$\Rightarrow$: Assume $P =_{ST} Q$. To establish that $P \sim Q$ it suffices to show that $=_{ST}$ is a strong bisimulation. Suppose $P \rightarrow^a P'$; therefore $<a, S(P')> \in S(P)$, and since $S(P) = S(Q)$ $<a, S(P')> \in S(Q)$, implying the existence of a Q' such that $Q \rightarrow^a Q'$ and $S(Q') = S(P')$, or $Q' =_{ST} P'$. Symmetry allows us to conclude that $=_{ST}$ is a strong bisimulation.

$\Leftarrow$: Assume $P \sim Q$. As $\sim$ is the weakest strong bisimulation on processes it follows that $S(P) \equiv S(Q)$ in the sense of section 5.1.1. Since set equality is strongly extensional because of AFA, it must be the case that $S(P) = S(Q)$.

□

The signigicance of this theorem lies in the fact that the equational axioms of $\sim$ are valid in ST; therefore, equational reasoning about $\sim$ can be mirrored as reasoning about $=_{ST}$.

5.4. Reasoning in ST

The remainder of the chapter examines several different ways in which, by using ST as a semantic basis, reasoning about processes may be conducted. The first method uses an algebraic style; equality in ST is exploited in the same way that strong equivalence is in the CCS formalism. The second way is more propositionally oriented; properties of programs are formulated as propositions in the predicate calculus, where the domain of individuals is ST. The third way involves a modal logic; using ST and the predicate calculus one gives a semantic account of the desired modal logic. The satisfaction relation defined for the logic then gives a means of specifying properties of programs, and the inference rules give means of proving properties.

The first two of these methods of deduction form the basis of the rest of this section, and the section following contains an extended example of the development of a modal logic in the style of the third method. With an account of ST in hand, the Nuprl system provides a convenient forum for discussing all three kinds of reasoning, since it provides facilities for reasoning about equalities and with the predicate calculus and for developing proof rules corresponding to the proof rules for alternative logics. Accordingly, references to Nuprl occur throughout the remainder of the chapter, and the next chapter details an actual development of a finitary subset of ST in Nuprl.

The remainder of this section contains examples of traditional synchronization problems worked out in CCS and in a CCS-like formalism.

Several of the problems have very natural algebraic specifications and can be easily expressed using the algebraic structure afforded by CCS. Other problems, however, are awkward to state algebraically but have a natural logic-based specification; problems like these can be handled in a straightforward fashion by quantifying in various ways over the elements of an object in *ST*. Still other problems are best attacked using a conjunction of these styles of reasoning. This section examines problems in each of these classes. Where appropriate, the solutions refer to aspects of the Nuprl system and logic that help in the statement or solution of a problem.

5.4.1. Reasoning Algebraically

This section describes algebraic specifications and proofs of correctness of standard synchronization problems and properties of concurrency in CCS. The first problem involves mutual exclusion, in which it is desired that two processes execute certain sections (called *critical* sections) of themselves without the other process being able to execute its analogous section. Binary semaphores solve this problem; the goal, then, is to develop and prove correct a CCS expression that implements a binary semaphore. (This example is due to Milner [Milner 80].) Suppose process P_1 has critical section $\alpha\beta$ and P_2 has critical section $\gamma\delta$, where α, β, γ and δ are atomic actions. Consider the following definitions:

$$P_1 = \pi\alpha\beta\underline{\phi}\, P_1$$

$$P_2 = \pi\gamma\delta\underline{\phi}\, P_2$$

$$Sem = \underline{\pi\phi}\, Sem$$

$$Q = (P_1|Sem|P_2) \setminus \pi \setminus \phi.$$

Intuitively π and ϕ correspond to the P and V operations on a semaphore, and *Sem* is a process that implements a semaphore. To see that mutual exclusion is enforced, consider the form of process Q. Using the expansion theorem of section 2.3.2 and the definition of restriction, it is easy to see that

$$Q \sim (\tau\alpha\beta\tau Q + \tau\gamma\delta\tau Q);$$

in other words, the critical sections of P_1 and P_2 are executed atomically.

In *ST*, this theorem boils down to proving set equality. Using the definition of Q and the definition of CCS composition given earlier, proving the theorem

$$\vdash Q =_{ST} (\tau\gamma\delta\tau Q + \tau\alpha\beta\tau Q)$$

establishes the correctness of *sem*; that is, the critical sections of P_1 and P_2 cannot be interleaved with each other.

Deadlock also has a natural algebraic definition in CCS. Intuitively, deadlock happens when two or more processes cannot execute and have not terminated; for simplicity we restrict our attention to the two-process case. To capture deadlock formally in CCS we first define a set of *synchronization* actions, $Syn \subseteq A$. *Syn* will typically contain events corresponding to *send* and *recieve* or *signal* and *wait*. Intuitively, the presence of these events causes the possibility of deadlock in that they cause processes to wait for an event to be executed by another process; without these actions circular waiting would not be possible. Processes P_1 and P_2 are deadlocked if it is the case that neither process is *NIL* and

$$(P_1|P_2)\backslash Syn \sim NIL.$$

This condition specifies deadlock because the process $(P_1|P_2) \backslash Syn$ allows no events in Syn to happen in isolation; these events can occur only in the context of synchronization with their inverses (with a τ resulting). Thus, if $(P_1|P_2) \backslash Syn \sim NIL$ then the only top-level actions to P_1 and P_2 are events in Syn which have no corresponding inverse actions in the other process with which to synchronize. The processes are therefore engaged in a circular wait. For example, consider the following, where $Syn = \{\alpha, \beta\}$.

$$P = \alpha\underline{\beta}NIL$$

$$Q = \beta\underline{\alpha}NIL$$

Processes P and Q are deadlocked because P is waiting to synchronize on α with Q while Q is waiting to synchronize on β with P. This is reflected in the fact that

$$(P|Q)\backslash Syn = NIL.$$

We can generalize this definition to an arbitrary number of processes in the obvious way.

Again the fact that equality in ST corresponds to $\sim$ in CCS enables the simplicity of the statement of the following theorem.

$$\vdash (P_1|P_2) \backslash Syn =_{ST} NIL$$

Facilities provided by a system like Nuprl for reasoning about substitution and equality would allow the proof to be conducted in a natural fashion.

The last example details a specification of the readers-writers problem, a problem which is paradigmatic of many database problems, and it illustrates a type of problem for which algebraic reasoning is unwieldy. The problem is this: given some number of processes that wish to *read* from a database and some number of processes that wish to *write* to a database, enforce the condition that at most one writer may be writing at any time and that any number of readers may be reading at any time, provided that no process is writing at that time.

In order to solve this problem it is convenient to extend the CCS formalism by generalizing the notion of prepending an action onto a process to the full notion of sequential execution. Let P and Q be processes, and define the process PQ operationally as follows.

$$Q \rightarrow^{b} Q' \quad \Rightarrow \quad NILQ \rightarrow^{b} Q'$$

$$P \rightarrow^{a} P' \quad \Rightarrow \quad PQ \rightarrow^{a} P'Q$$

The ST representation of this can be defined as follows. First, for elements s and t of ST define the operation *sequence.*

$$\begin{aligned} sequence(s, t) &= t \qquad \text{if } s = \emptyset \\ &= \{<a,\ sequence\ (s',\ t)> \mid <a,\ s'> \in s\ \&\ s' \neq \emptyset\} \\ &\qquad \text{otherwise} \end{aligned}$$

Clearly $sequence(s, t)$ is an element of ST. Now,

$$S(PQ) = sequence(S(P), S(Q)).$$

The correctness of this is easily checked.

To solve the readers-writers problem, we will define a process *sched* which enforces the readers-writers condition, provided that readers issue β_r (for "begin read") before each read operation and ϕ_r (for "finish read") after each read and that writers issue corresponding β_w and ϕ_w before and after their writes. We define reads and writes to consist of two atomic actions, r_b and r_f in the case of reads (for "read begin" and "read finish") and w_b and w_f in the case of writes; the necessity of this convention stems from the fact that in the solution the β- and ϕ-actions will synchronize with their inverses with τ's resulting, leaving no means of referring to active readers or active writers. Consider the following definitions.

$$writer_i = \beta_w\, w_b w_f\, \phi_w\, writer_i$$

$$reader_i = \beta_r\, r_b r_f\, \phi_r\, reader_i$$

$$sched = (\underline{\beta}_w\, \underline{\phi}_w + R)\, sched, \text{ where}$$

$$R = \underline{\beta}_r \underline{\phi}_r + \underline{\beta}_r R' \underline{\phi}_r$$

$$R' = \underline{\beta}_r \underline{\phi}_r R' + \underline{\beta}_r R \underline{\phi}_r .$$

A solution of the readers-writers problem would then be

$$(writer_1 \mid \ldots \mid writer_n \mid sched \mid reader_m \mid \ldots \mid reader_1) \backslash\beta_r \backslash\beta_w \backslash\phi_r \backslash\phi_w .$$

Arguing for the correctness of this implementation is very hard to do algebraically; it is most naturally carried out in another deductive framework. Informally one can argue that this is correct by noting that *sched* allows a writer to write exactly when there are no other writers writing or readers reading and that *sched* permits reading only if no writes are occurring, with any number of reads occurring simultaneously. This sort of argument is unconvincing; in the next section we shall see how one may use the predicate calculus in conjunction with ST to state and prove formally properties of processes.

5.4.2. Reasoning Logically

Some properties of concurrent programs have natural expressions as equations; an example of such a property is deadlock, as mentioned above. Other properties, however, do not have obvious statements involving only equivalence; examples of such properties include the readers-writers property and deadlock-freedom, where we wish to say that no subprocesses of two processes may deadlock. Both of these properties, however, do have natural propositional statements, and the deductive apparatus associated with the predicate calculus enable these properties to be proven of programs. Consider first the readers-writers problem alluded to in the previous section. What follows is a general framework for reasoning about the readers-writers problem using the Nuprl logic; it is used to suggest the correctness of the CCS-like solution presented earlier.

We first introduce a couple of intermediary notions to make what follows more concise and understandable. In order to discuss the notions of "active readers" and "active writers" it is convenient to define the traces of a process. The traces corresponding to a member P of ST may be defined as follows.

$$traces(P) = \{s \in A^* \mid s{=}nil \vee \exists y{\in}t.\ head(s){=}first(y)\ \&\ tail(s) \in traces(second(y))\}$$

That is, for a given object P in ST, $Traces(P)$ is a set whose elements are lists of actions (traces) that the process corresponding to t may perform, starting from its initial state. An actual Nuprl term computing *Traces* would be a function mapping ST to the collection U_1 of types; for the type of finitary elements of ST this function could be defined recursively using a *rec_ind* form as follows.

$$Traces = \lambda P. \quad rec\_ind(P; traces, Q.$$
$$\{s{:}A\ list \mid s{=}nil\ in\ A\ list \vee$$

$\exists y \in Q.\ head(s) = y.1 \ in \ A \ \& \ \exists z{:}traces(y.2).\ tail(s) = z$
$in \ A \ list\}$

Now, for any given element *rw* of *ST* that corresponds to a solution to the readers-writers problem, we assume the existence of functions *ar* and *aw* that for any trace of the solution compute the number of active readers and writers; such functions must exist, for otherwise reasoning about the solution would be impossible. In the solution presented above, *ar* has the following definition.

$$
\begin{aligned}
ar &: Traces(rw) \rightarrow int \\
&= \lambda s.\, if\ s = nil\ then\ 0\ else\ \ if\ hd(s) = r_b\ then\ 1 + ar(tl(s)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad else\ if\ hd(s) = r_f\ then\ -1 + ar(tl(s)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad else\ ar(tl(s))
\end{aligned}
$$

If the trace is empty then there are no active writers. If the first action in the trace is a "begin read" (r_b) then a process has begun a read operation and the number of active readers in the trace is incremented by one; if, on the other hand, the first action is a "finish read" then a process has completed a read operation, and the number of active readers in the trace is decremented by one. The function *aw* has a completely symmetric definition; its definition is therefore omitted.

In Nuprl the following term computes *ar*, assuming (for simplicity) that atomic actions are represented by integers.

$$
\begin{aligned}
ar = \ & \lambda s.list\_ind(s;\, 0;\, hd, tl, ind. \\
& \quad int\_eq(hd;\, r_b;\quad 1 + ind;\quad int\_eq(hd;\, r_f;\, -1 + ind;\, ind))))
\end{aligned}
$$

The three expressions in the *list_ind* form correspond to the term on which list induction is being performed, the result (*0*) to be returned if the term is the empty list, and the result to be returned if the term is *hd.tl*, where *hd* represents the head of the list, *tl* represents the tail of the list, and *ind* designates the recursive invocation of the *list_ind* form on term *tl*. The first two of the four places in the *int_eq* term designate the terms for which equality is being determined; the next two represent the *then* and *else* terms, respectively.

A process *rw* solves the readers-writers problem if the following theorem is provable.

$$\vdash \forall t \in Traces(rw).(aw(t)=0 \wedge ar(t) \geq 0) \vee (aw(t)=1 \wedge ar(t)=0)$$

This statement is provable of the CCS-like solution alluded to previously.

The remainder of this section examines deadlock-freedom as an example of a property best discussed using a combination of algebraic and logic-based reasoning. Intuitively, two processes are *deadlock-free* if it is impossible for them to evolve into a state in which they are deadlocked. To define this property in terms of the model it is convenient to define the notion of a subprocess of an element of ST. Using the definition of $\rightarrow^s$ given chapter 4, a process P' of a process P may be defined as follows.

P' is a *subprocess* of P if and only if $\exists s \in A^*.P \rightarrow^s P'$.

A *nontrivial* subprocess P' of P is a subprocess which is not NIL. Given these definitions, and given that Syn is defined as before, the following expresses deadlock-freedom with respect to Syn.

P and Q are *deadlock-free* exactly when

$\forall P',Q':ST.$ P' a nontrivial subprocess of P and Q' a nontrivial subprocess of $Q \Rightarrow (P'|Q') \setminus Syn \neq NIL$.

Proving this property for specific P and Q would entail proving that for arbitrary P' and Q' that are nontrivial subprocesses of P and Q, respectively, their composition, when restricted by Syn, is nontrivial. Typically a proof of this would involve a case analysis on the structure of the possible subprocesses of P and Q, and as the number of such subprocesses can be large using a system like Nuprl that can keep track of the subcases and also provide a facility for proving most of the trivial cases automatically would be invaluable.

5.5. A Development of a Modal Logic in the Model

Another proof style that is often used to prove properties of concurrent programs involves the use of a modal logic. Properties can be stated as formulas in the modal logic, and the inference rules of the modal logic can be used to prove that specific programs have the properties. This section details the development of a subset of the Hennessy-Milner Logic using Nuprl as a metalanguage in conjuction with finitary ST. The point of doing this is to illustrate the flexibility that the model affords in the style of reasoning one can perform and to point up the advantages that developing a model like ST in a Nuprl-like system affords a logic implementor.

5.5.1. The HM Logic

HM logic is a modal logic. The atomic proposition is T (for "true"), and in addition to the usual propositional connectives the logic provides modal operators indexed by CCS actions for building propositions. Formally, propositions may be described syntactically as follows:

$$\phi ::= T \mid \neg\phi \mid \phi\wedge\phi \mid <a>\phi$$

where a is an action. Semantically the propositional connectives are interpreted in the usual sense; we may also define derived propositions and connectives such as F, $\wedge$ and $\Rightarrow$. Intuitively, a process satisfies $<a>\phi$ if the process can execute an a action and the resulting subprocess satisfies ϕ. We can define the dual modal operator $[a]$ as $[a]\phi = \neg <a> \neg\phi$; a process satisfies $[a]\phi$ if whenever it performs an a action the resulting subprocess satisfies ϕ. To specify the semantics formally, we define a satisfaction relation inductively on processes and predicates.

$P \models T$	for all processes P.
$P \models \neg\phi$	if it is not the case that $P \models \phi$.
$P \models \phi_1 \wedge \phi_2$	if $P \models \phi_1$ and $P \models \phi_2$.
$P \models <a>\phi$	if and only if $\exists P'.\ P \rightarrow^a P'\ \&\ P' \models \phi$

5.5.2. A Type-Theoretic Semantics of HM using *ST*

In order to proceed with a type-theoretic account of HM we shall define the semantics of the HM propositions in terms of type inhabitation in the Nuprl theory. When the predicate calculus is modeled in type theory propositions are identified with types, and the satisfaction relation on propositions is identified with type inhabitation; a proposition is true when the corresponding type has an element in it. We adopt the same philosophy of identifying propositions with types, with one important modification. In the predicate calculus, the satisfaction relation is unary-- $\models$ is defined over formulas only. In the case of HM, however, $\models$ is binary; it is defined over processes and formulas. Accordingly, instead of modeling HM formulas as types we shall model them as type-valued functions over *ST*, and the clause $P \models \phi$ will be interpreted as inhabitation of $N(\phi)(P)$, where $N(\phi)$ is Nuprl the translation of ϕ.

We now consider the specific HM propositional constructors type-theoretically. Consider the case of *T*. In HM any process *P* satisfies *T*. Therefore, $N(T)$ should be inhabited for any *P*; $\lambda P.(0\ in\ int)$ is such a type-valued function. In the case of $\neg\phi$, the translation should contain an element for process *P* exactly when $N(\phi)$ is empty; accordingly, $N(\neg\phi) = \lambda P.(N(\phi)(P) \rightarrow void)$.

Now consider $\phi_1 \wedge \phi_2$. A processs *P* satisfies $\phi_1 \wedge \phi_2$ when it satisfies each conjunct. The correspond type-valued function, then, must be inhabited exactly when $N(\phi_1)(P)$ and $N(\phi_2)(P)$ are; accordingly, let

$$N(\phi_1 \wedge \phi_2) = \lambda P.(N(\phi_1)(P) \# N(\phi_2)(P)).$$

Finally, consider the case of $<a>\phi$. In *ST* we have interpreted $P \rightarrow^{a} P'$ and $<a, P'> \in P$, so, using the fact that $\exists$ and & can be modeled (see chapter 3), $<a>\phi$ may be modeled as

$$N(<a>\phi) = \lambda P. \downarrow \exists P' \in P.\ <a, P'> \in P\ \&\ N(\phi)(P'),$$

where for a type T $\downarrow T$ is the type $\{0\ in\ int \mid T\}$. The necessity for $\downarrow$ stems from the fact that the account of $\exists$ is constructive in that the proof of a $\exists$-formula must specify how to build a witness to the truth of the body of the formula. Since the semantics of HM are not constructive, however, we do not want to specify the construction of an object, and "squashing" the $\exists$ in this way accomplishes this goal. $\downarrow \exists P' \in P.\ <a, P'> \in P\ \&\ N(\phi)(P')$ is inhabited by *axiom* whenever $\exists P' \in P.\ <a, P'> \in P\ \&\ N(\phi)(P')$ is inhabited, it should be noted.

5.5.3. HM Proofs in Nuprl

The rest of this section describes a Nuprl account of the proof system of a subset of HM due to Stirling [Stirling 85a]. We shall show that his axioms are all provable in Nuprl and that his rules of inference, when restricted to finitary *ST*, correspond to derived rules of inference in the Nuprl system. Stirling's system allows one to reason about HM formulas in the context of SCCS expressions; that is, the synchronous product, $\otimes$, is used as a constructor, and the action set A is an Abelian group with function $\circ$. He also disallows negation; the subset of HM he considers is the following.

$$\phi ::= T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi \mid <a>\phi \mid [a]\phi \quad \text{where } a \in A$$

The Nuprl translations of these formulas are given in table 5.1. The SCCS expressions allowed are the closed expressions defined by the following grammar.

$$P ::= Z \mid NIL \mid aP \mid fix\ Z.P \mid P+P \mid P \otimes P \quad \text{where } a \in A$$

Z denotes a variable. Since we are restricting ourselves to finitary *ST*, it is impossible to consider nonterminating processes. Therefore, we shall not consider the operator *fix* in what follows.

In the remainder of the chapter, Greek letters will refer to HM formulas (and their translations in Nuprl), italic Roman letters from the beginning of

Table 5.1. Nuprl Translations of Stirling's HM Formulas. $N(A)$ represents the Nuprl translation of formula A.

HM Formula	Nuprl Translation
T	$\lambda P.(0\ in\ int)$
F	$\lambda P.void$
$\phi_1 \wedge \phi_2$	$\lambda P.N(\phi_1)(P)\#N(\phi_2)(P)$
$\phi_1 \vee \phi_2$	$\lambda P.N(\phi_1)(P) \vert N(\phi_2)(P)$
$<a>\phi$	$\lambda P.\downarrow \exists y \in P.\ y.1 = a\ in\ A \wedge N(\phi)(y.2)$
$[a]\phi$	$\lambda P.\downarrow \forall y \in P.\ y.1 = a\ in\ A \wedge N(\phi)(y.2)$

the alphabet will refer to elements of A, and italic Roman letters $P, Q, R, \ldots$ will refer to SCCS processes (or their ST equivalents).

Stirling defines two notions of provability: $\vdash$ and $\vdash_\Phi$, where ϕ is a HM formula. $\vdash$ has the usual meaning; $P \vdash \phi$ holds if it can be deduced from the axioms and inference rules that P satisfies ϕ. $\vdash_\Phi$, on the other hand, has a different meaning; $P \vdash_\Phi \Psi$ if for every Q such that $Q \vdash \phi$, $P \otimes Q \vdash \Psi$. It is easy to check that $\vdash_\Phi$ is compositional: if $P \vdash_\Phi \Psi$ and $Q \vdash_\Psi \Gamma$ then $P \otimes Q \vdash_\Phi \Gamma$.

In the Nuprl account, $P \vdash \phi$ will be interpreted as $\vdash_{Nuprl} N(\phi)(P)$; in general, we shall not distinguish between $\vdash$ and $\vdash_{Nuprl}$ when our intent is clear from context. Likewise, $P \vdash_\Phi \Psi$ will correspond to the following.

$$\vdash_{Nuprl} \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ \Psi(P \otimes Q)$$

5.5.3.1. Axioms

The axioms of Stirling's system will now be considered. Table 5.2

Table 5.2. Stirling's Axioms for HM.

Axiom	Translation
$P \vdash T$	$\vdash$ *0 in int*
$NIL \vdash [a]\phi$	$\vdash \forall y \in \varnothing.\, y.1 = a \text{ in } A \Rightarrow \phi(y.2)$
$P \vdash_{\phi} T$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ 0 \text{ in int}$
$P \vdash_{F} \phi$	$\vdash \forall Q{:}\{t{:}ST \mid void\}.\ \phi(p \otimes q)$
$NIL \vdash_{\phi} [a]\Psi$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ \forall P' \in NIL \otimes Q.\, y.1 = a \text{ in } A \Rightarrow \Psi(y.2)$
$aP \vdash [b]\phi$ if $b \neq a$	$\vdash \forall a{:}A.\ b \neq a \Rightarrow \forall P' \in aP.\, P'.1 = b \text{ in } A \Rightarrow \phi(P'.2)$
$aP \vdash_{\phi} [b]\Psi$ if b\a does not exist	$\vdash \forall a{:}A.\ \forall Q{:}\{t{:}ST \mid A(t)\}.\forall P' \in aP \otimes Q.\, P'.1 = b \text{ in } A \Rightarrow \Psi(P'.2)$ assuming that b\a does not exist

summarizes the axioms of Stirling's system and the Nuprl translations of them; in this table, and in the rest of the chapter β-redexes will be expanded in the Nuprl representations. For the sake of readability we shall also suppress $\downarrow$ when it appears. We should note that if $c = b \circ a$ then $b \backslash c = a$.

All the axioms, in their Nuprl form, are easily proved. $\vdash$ *0 in int* follows immediately from equality reasoning, $\vdash \forall y \in \varnothing . y.1 = a \text{ in } A \Rightarrow \phi(y.2)$ follows from the fact that ∀-intro on an empty collection allows one to conclude the truth of the body of the formula at once, and $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.0\ in\ int$ follows from an $\forall$-intro and the provability of $0\ in\ int$. $\vdash \forall Q{:}\{t{:}ST \mid void\}.\ \phi(P \otimes Q)$ is provable using $\forall$-intro since $\{t{:}ST \mid void\}$ is provably empty. $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ \forall P'{:}NIL \otimes Q.\ y.1 = a\ in\ A \Rightarrow \Psi(y.2)$ follows from two applications of $\forall$-intro and the easily proved observation that $NIL \otimes Q = NIL\ in\ ST$ and hence $NIL \otimes Q$ is empty. $\vdash \forall a{:}A.\ b \neq a \Rightarrow \forall P' \in aP.\ P'.1 = b \Rightarrow \phi(P'.2)$ can be proved by performing an $\forall$-intro and two $\Rightarrow$-intros and observing that since aP has only one member, and that member does not have b as its first element, then $\phi(P'.2)$ vacuously holds. Finally, if $b \backslash a$ does not exist then $\vdash \forall a{:}A.\ \forall Q{:}\{t{:}ST|\phi(t)\}.\forall P' \in aP \otimes Q.\ P'.1 = b\ in\ A \Rightarrow \Psi(P'.2)$ follows from two $\forall$-intros and the observation, easily proven from the properties of $\otimes$, that $aP \otimes Q$ is an empty collection if $b \backslash a$ does not exist.

5.5.3.2. The Inference Rules

The rest of the section describes the inference rules. Stirling's rules are divided into eight categories; in our presentation we shall analyze the rules category by category. In each case a table will contain a translation of Stirling's rules into the corresponding Nuprl theorems; in the discussion following the table remarks about the validity of these theorems in Nuprl will be made, and general descriptions of tactics that could be written to implement these rules will be given. We take the liberty of rewriting Stirling's rules in a refinement (i.e., top-down) style, since Nuprl rules are refinement rules.

Table 5.3 lists the $\vee I$ (for $\vee$-introduction) inference rules. The first two

Table 5.3. Stirlings's $\bigvee I$ Rules.

Inference Rule	Translation
$P \vdash \phi \vee \Psi$ $P \vdash \phi$	$\vdash \phi(P) \mid \Psi(P)$ $\vdash \phi(P)$
$P \vdash \phi \vee \Psi$ $P \vdash \Psi$	$\vdash \phi(P) \mid \Psi(P)$ $\vdash \Psi(P)$
$P \vdash_{\phi} \Psi \vee \Gamma$ $P \vdash_{\phi} \Psi$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\Psi(P \otimes Q) \mid \Gamma(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\Psi(P \otimes Q)$
$P \vdash_{\phi} \Psi \vee \Gamma$ $P \vdash_{\phi} \Gamma$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\Psi(P \otimes Q) \mid \Gamma(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\Gamma(P \otimes Q)$
$P \vdash_{\phi \vee \psi} \Gamma$ $P \vdash_{\phi} \Gamma$ $P \vdash_{\psi} \Gamma$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(Q) \mid \Psi(Q)\}.\Gamma(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \phi(Q)\}.\Gamma(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \Psi(Q)\}.\Gamma(P \otimes Q)$

rules correspond to the $|$-introduction rules in the Nuprl logic. The next three rules have straightforward implementations as *tactics*, or derived rules of inference, in the Nuprl system. We shall describe informally the implementation of the last rule. Assuming the proofs of the subgoal, the first step in proving the goal involves a step of $\forall$-introduction; the existence of an arbitrary Q of the appropriate type is assumed, and we know that $\phi(Q) \mid \Psi(Q)$ is true. If $\phi(Q)$ is true then then proof of the first subgoal (via a step of $\forall$-elimination) allows one to conclude that $\Gamma(P \otimes Q)$ holds; similarly, if $\Psi(Q)$ is true then the proof of the second subgoal gives the conclusion.

Table 5.4 lists the $\wedge I$ (for $\wedge$-introduction) rules. The first rule

Table 5.4. Stirling's $\wedge I$ Rules.

Inference rule	Translation
$P \vdash \phi \wedge \Psi$ $P \vdash \phi$ $P \vdash \Psi$	$\vdash \phi(P) \wedge \Psi(P)$ $\vdash \phi(P)$ $\vdash \Psi(P)$
$P \vdash_{\phi \wedge \psi} \Gamma$ $P \vdash_{\phi} \Gamma$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(t) \wedge \Psi(t)\}.\ \Gamma(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ \Gamma(P \otimes Q)$
$P \vdash_{\phi \wedge \psi} \Gamma$ $P \vdash_{\psi} \Gamma$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(t) \wedge \Psi(t)\}.\ \Gamma(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \Psi(t)\}.\ \Gamma(P \otimes Q)$
$P \vdash_{\phi} \Psi \wedge \Gamma$ $P \vdash_{\phi} \Psi$ $P \vdash_{\phi} \Gamma$	$\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ \Psi(P \otimes Q) \wedge \Gamma(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ \Psi(P \otimes Q)$ $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\ \Gamma(P \otimes Q)$

corresponds to the $\wedge$-introduction rule in the logic. The last three rules involve simple sequences of $\forall$-introduction and $\wedge$-introduction and elimination rules and may be coded as tactics. To implement the second rule, for instance, assume the proof of the subgoal and perform an $\forall$-introduction on the goal. Using the $Q{:}\{t{:}ST \mid \phi(t) \wedge \Psi(t)\}$ assumption resulting from the introduction rule, perform an $\forall$-elimination using the subgoal to conclude that $C(P \otimes Q)$ holds.

Table 5.5 lists the $<a>$ introduction rules. The first rule is simple $\exists$-

Table 5.5. Stirling's $<a>I$ Rules.

Inference rule	Translation
$aP \vdash <a>\phi$ $P \vdash \phi$	$\vdash \exists y \in aP.y.1 = a \textit{ in } A \wedge \phi(y.2)$ $\vdash \phi(P)$
$bP \vdash {}_{<a\backslash b>\phi} <a>\Psi$ $P \vdash_{\phi} \Psi$ assuming a\b exists	$\vdash \forall Q{:}\{t{:}ST \mid \exists y \in t.y.1 = c \textit{ in } A \wedge$ $\phi(y.2)\}.\ \exists y \in bP \otimes Q.\ y.1 = a \textit{ in } A \wedge$ $B(y.2)$ $\vdash a = b \circ c$ $\vdash \forall Q'{:}\{t{:}ST \mid \phi(t)\}.\Psi(P \otimes Q')$

intro in the logic. The second can be implemented with a tactic which performs the following Nuprl reasoning. Given the truth of the subgoals, the tactic first performs an $\forall$-introduction on the goal. From the definition of $\otimes$ the tactic concludes that $(bP)\otimes Q$, where Q is the just-introduced member of $\{t{:}ST \mid \exists y \in t.y.1 = c \textit{ in } A \;\&\; \phi(y.2)\}$, comprises one element, a pair whose first element is a and whose second element is equal to $P \otimes Q'$, where Q' satisfies ϕ. From the second subgoal, the tactic therefore concludes that the second element of the pair must satisfy Ψ.

Table 5.6 lists the introduction rules for *[a]* formulas. The first rule has

Table 5.6. Stirling's *[a]I* Rules.

Inference rule	Translation
$aP \vdash [a]\phi$ $P \vdash \phi$	$\vdash \forall y \in aP.y.1 = a \text{ in } A \wedge \phi(y.2)$ $\vdash \phi(P)$
$bP \vdash_{[a\backslash b]\phi} [a]\Psi$ $P \vdash_{\phi} \Psi$ assuming $a\backslash b$ exists	$\vdash \forall Q{:}\{t{:}ST \mid \forall y \in t.y.1 = c \text{ in } A \Rightarrow$ $\phi(y.2)\}.\forall y \in bP \otimes Q.\, y.1 = a \text{ in } A$ $\Rightarrow \Psi(y.2)$ $\vdash a = b \circ c \text{ in } A$ $\vdash \forall Q{:}\{t{:}ST \mid \phi(t)\}.\Psi(P \otimes Q)$

a straightforward implementation, given the definition of aP. The second can be implemented as a tactic in the following fashion. Assuming the truth of the subgoals, the tactic first introduces an arbitrary Q of type $\{t{:}ST \mid \forall y \in t.y.1 = c \text{ in } A \Rightarrow \phi(y.2)\}$; from this it knows that each pair in Q having c as its first element has a second element satisfying ϕ. From the definition of bP and $\otimes$, then, the tactic may conclude that each pair in $(bP) \otimes Q$ having a $(= b \circ c)$ as its first element must have a second element of the form $P \otimes Q'$, where Q' satisfies ϕ. The second subgoal therefore allows the conclusion of $\Psi(P \otimes Q')$.

Table 5.7 lists the $+<>I$ rules. The first two rules can be implemented

Table 5.7 Stirling's $+<>I$ Rules.

Inference rule	Translation
$P+Q \vdash <a>\phi$ $P \vdash <a>\phi$	$\vdash \exists y \in P+Q. y.1=a \textit{ in } A \wedge \phi(y.2)$ $\vdash \exists y \in P. y.1=a \textit{ in } A \wedge \phi(y.2)$
$P+Q \vdash <a>\phi$ $Q \vdash <a>\phi$	$\vdash \exists y \in P+Q. y.1=a \textit{ in } A \wedge \phi(y.2)$ $\vdash \exists y \in Q. y.1=a \textit{ in } A \wedge \phi(y.2)$
$P+Q \vdash_{\phi} <a>\Psi$ $P \vdash_{\phi} <a>\Psi$	$\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}. \exists y \in (P+Q) \otimes R.$ $y.1=a \textit{ in } A \wedge \Psi(y.2)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}. \exists y \in P \otimes R.$ $y.1=a \textit{ in } A \wedge \Psi(y.2)$
$P+Q \vdash_{\phi} <a>\Psi$ $Q \vdash_{\phi} <a>\Psi$	$\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}. \exists y \in (P+Q) \otimes R.$ $y.1=a \textit{ in } A \wedge \Psi(y.2)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}. \exists y \in Q \otimes R.$ $y.1=a \textit{ in } A \wedge \Psi(y.2)$

with very simple tactics which use the definition of $P+Q$ to achieve the desired result. Likewise the third and fourth rules have relatively straightforward implementations based upon an analysis of $+$ and $\otimes$. A tactic implementing the third rule, for example, would first perform an $\forall$-intro to get an arbitrary R in $\{t{:}ST \mid \phi(t)\}$. Using the assumed proof of the subgoal, the tactic would then conclude that $\exists y \in Q \otimes R.\ y.1=a \textit{ in } A \wedge \Psi(y.2)$, and from the easily proven fact that $(P+Q) \otimes R = (P \otimes R) + (Q \otimes R) \textit{ in } ST$ it may then deduce that the y guaranteed by the subgoal is the y needed to establish the truth of the goal.

Table 5.8 lists the $+[]$ introduction rules. These rules have a

Table 5.8. Stirling's $+[]I$ Rules.

Inference rule	Translation
$P+Q \vdash [a]\phi$ $P \vdash [a]\phi$ $Q \vdash [a]\phi$	$\vdash \forall y \in P+Q.\ y.1=a \text{ in } A \Rightarrow \phi(y.2)$ $\vdash \forall y \in P.\ y.1=a \text{ in } A \Rightarrow \phi(y.2)$ $\vdash \forall y \in Q.y.1=a \text{ in } A \Rightarrow \phi(y.2)$
$P+Q \vdash_{\phi} [a]\Psi$ $P \vdash_{\phi} [a]\Psi$ $Q \vdash_{\phi} [a]\Psi$	$\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\forall y \in (P+Q) \otimes R.$ $y.1=a \text{ in } A \Rightarrow \Psi(y.2)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\forall y \in P \otimes R.$ $y.1=a \text{ in } A \Rightarrow \Psi(y.2)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\forall y \in Q \otimes R.$ $y.1=a \text{ in } A \Rightarrow \Psi(y.2)$

straightforward encoding as tactics. A tactic for the first rule, for instance, would first perform an $\forall$-intro to get an arbitrary y in $P+Q$; from proofs of the subgoals and from the description of $+$, the tactic concludes that y must either be in P or Q and that the main goal must therefore hold. The implementation of the second rule is similar.

Table 5.9 lists the introduction rules for $\otimes$. The first and second rules correspond to the $\forall$-elimination rule in Nuprl, using P (in the first rule) and Q (in the second rule) as the object of elimination. The third rule can be implemented as a tactic which first performs $\forall$-introduction on the goal to obtain R in $\{t{:}ST \mid \phi(t)\}$ followed by an $\forall$-elimination on the first subgoal using R and an $\forall$-elimination on the second subgoal using $P \otimes R$. The fourth rule is symmetric with the third rule.

Table 5.9. Stirling's $\otimes I$ Rules.

Inference rule	Translation
$P\otimes Q \vdash \Psi$ $P \vdash \phi$ $Q \vdash_{\phi} \Psi$	$\vdash \Psi(P\otimes Q)$ $\vdash \phi(P)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\ \Psi(Q\otimes R)$
$Q\otimes P \vdash \Psi$ $P \vdash \phi$ $Q \vdash_{\phi} \Psi$	$\vdash \Psi(Q\otimes P)$ $\vdash \phi(P)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\ \Psi(P\otimes R)$
$P\otimes Q \vdash_{\phi} \Gamma$ $P \vdash_{\phi} \Psi$ $Q \vdash_{\Psi} \Gamma$	$\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\ \Gamma(P\otimes Q\otimes R)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\ \Psi(P\otimes R)$ $\vdash \forall R{:}\{t{:}ST \mid \Psi(t)\}.\ \Gamma(Q\otimes R)$
$Q\otimes P \vdash_{\phi} \Gamma$ $P \vdash_{\phi} \Psi$ $Q \vdash_{\Psi} \Gamma$	$\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\Gamma(Q\otimes P\otimes R)$ $\vdash \forall R{:}\{t{:}ST \mid \phi(t)\}.\ \Psi(P\otimes R)$ $\vdash \forall R{:}\{t{:}ST \mid \Psi(t)\}.\ \Gamma(Q\otimes R)$

Stirling also presents induction-based rules for reasoning about the *fix* operator. Since we have restricted ourselves to terminating CCS expressions by considering only finitary *ST*, we cannot give an account of these rules. We are presently working on a model which admits nonterminating processes; this model is based on *lazy* types [Mendler, Panangaden and Constable 86], and the induction principle associated with these types appears to correspond to Stirling's induction principle.

5.6 Conclusion

This chapter has presented a model of concurrency that is expressive enough to model CCS; we have also attempted to show that the model in conjunction with a logic like Nuprl is expressive enough to enable several different modes of logical reasoning to be pursued.

Mention was also made on occasion of the benefits to be gained by using a system like Nuprl to develop an environment for reasoning in a machine-assisted fashion about concurrency. The next chapter contains a description of an actual implementation carried out in the system of some of what was discussed in this chapter.

Chapter 6

Implementing CCS

The preceding chapter described how in principle one can reason about concurrency in general, and CCS in particular, by using the *ST* model in the context of Nuprl. We have also claimed that this wedding of *ST* and Nuprl represents a reasonable framework for developing an automated proof assistant for proving properties about concurrent processes. In order to buttress this claim somewhat this chapter describes an actual implementation in the Nuprl proof development system of some of the ideas developed earlier in the thesis.

6.1. Introduction

This chapter describes a Nuprl library containing a development of the finitary *ST* model of CCS. The ultimate goal of the implementation is to prove that the binary semaphore implementation given in chapter 5 is correct, and in the course of doing so we also prove the expansion theorem for CCS that appears in chapter 2.

The library itself contains 240 Nuprl objects, where an object is either a definition, a theorem, a collection of bindings for use by the evaluator, or a collection of ML programs and bindings that define operations to be used by the metalanguage in developing derived proof rules. The first 50 or so of these objects comprise definitions of standard logical connectives and quantifiers in terms of the underlying type theory. The next 150 objects develop the finite set theory necessary to give an account of the *ST* model, and the last 40 objects present a development of CCS. The appendix at the end of the thesis contains a listing of the library.

The implementation makes frequent use of a convention for defining new type-theoretic objects that we shall briefly describe here. The Nuprl type theory is not decidable, meaning that in general it is impossible to determine automatically whether a term has a given type. Furthermore, there is no means in the Nuprl definition mechanism for describing the type of the defined notation; therefore, every time a definition is invoked, its type must be re-established. In order to stem somewhat the consequent proliferation of typing subgoals (subgoals of the form "t in T" for term t and type term T) in proofs of theorems involving defined objects, we will frequently introduce such an object (like *finset*) by proving a theorem, named by the object followed by an underscore (i.e., *finset_*), whose statement is the type of the object being defined and whose first proof step involves explicitly introducing the object's definition. The statement of *finset_*, for example, is $U_1 \rightarrow U_1$, and the first step of the proof explicitly introduces the Nuprl term having type $U_1 \rightarrow U_1$ that we wish to correspond to *finset*. To define *finset*, then, one creates a Nuprl definition object whose right-hand side is the term extracted from *finset_*. The system understands this convention, and whenever it sees the notation *finset* it can deduce that *finset* has type $U_1 \rightarrow U_1$. Although this methodology forces the polymorphism in operators to be explicit--all the set-theoretic operations, for example, have an explicit type parameter that does not usually appear in the body of the definition of the operation--it enables well-formedness subgoals to be caught and induces a concise term structure. This last is a very desirable property, given the somewhat inefficient term-handling that Nuprl provides.

The rest of the chapter is organized in the following fashion. Sections 6.2 and 6.3 describe the implementation of *finset* and *ST*, and CCS in *ST*, respectively. Section 6.4 then discusses the observations and insights acquired from using Nuprl to develop an extensive theory.

6.2. Implementing Finite Sets and the *ST* Model

As we saw in the previous chapter, the model *ST* rests on a set-theoretic foundation. Since Nuprl supplies type theory as a foundational langauge, and since types are not sets, the first stage in developing *ST* involves implementing finite set theory in Nuprl. In particular, one should have available a type of finite sets as well as traditional set operations such as union, collection and comprehension and basic sets like the null set and singleton sets. One should also be able to appeal to the axioms that define the semantics of these set-theoretic objects. Accordingly, in our implementation of finite sets we define set-theoretic objects like union and then prove that the associated axioms in set theory are true of our implementation.

The remainder of this section describes the implementation of finite sets in more detail and closes with an account of the development of ST.

6.2.1. The "Type Constructor" *finset*

Representing sets in a natural and efficient way, as well as a way that allows sets to be defined recursively, requires some thought in Nuprl. Identifying sets with their characteristic functions in the obvious fashion, for example, allows very straightforward implementations of set-theoretic functions and predicates using existing type constructors, as we saw in chapter 4. However, the positivity requirements that the implementation of the recursive type constructor enforce prohibit this sort of implementation. On the other hand, identifying sets with lists of their elements introduces inefficiencies in implementations of certain set functions like union.

We opt to represent sets as arrays with an appropriate equivalence relation defined on top of them. In Nuprl arrays over a type T may be defined as pairs, where the first element is an integer and the second element is a function from the initial segment of the positive integers defined by the first element of the pair into type T. The set equivalence relation then specifies that two arrays are equal as sets when they contain the same elements.

With this informal overview of how we implement sets we now briefly describe the library objects that we defined in order to model a type of finite sets for an arbitrary U_1 (or "small") type T. Table 6.1 lists the Nuprl definitions described in this section, and table 6.2 lists the associated typing

Table 6.1. Nuprl Definitions.

Name	Display Form	Body
p1	*<p>.1*	*spread(<p>; u,v.u)*
p2	*<p>.2*	*spread(<p>; u,v.v)*
Interval	*{1..<n>}*	*term__of(Interval__)(<n>)*
set__eq	*<s> =e <t> over <T>*	*term__of(set__eq__)(<T>)(<s>)(<t>)*
finset	*finset(<T>)*	*term__of(finset__)(<T>)*

Table 6.2. Nuprl Theorems.

Name	Statement
Interval__	$int \to U_1$
set__eq__	$T{:}U_1 \to T \to T \to U_1$
finset__	$U_1 \to U_1$

theorems. It should be noted that Nuprl definitions consist of a display form, which is what appears on the screen when a user invokes a definition, and a body, which defines the meaning of the display form. Parameters to the display form are surrounded by angle brackets. The definitions *p1* and *p2* define the notations *p.1* and *p.2* for arbitrary pairs *p* as Nuprl terms corresponding to the first and second projection functions for pairs, respectively. *Interval__* is a theorem that proves the well-formedness of a function that takes an integer and returns a subtype of the integers corresponding to the initial segment of the positive integers defined by the integer. That is,

$$term\_\_of(Interval\_\_) = \lambda n.\{x{:}int \mid 0<x\leq i\}.$$

Interval, then, is the Nuprl definition object that defines the notation *{1..n}* to be *term__of(Interval__)(n)*. For an arbitrary small type *T*, arrays of type *T* will be defined by the type *n:int#{1..n}* $\rightarrow$*T*, which defines the type of pairs described informally above.

The theorem *set__eq__* proves the well-formedness of the set equivalence for any small type *T* and any two arrays of type *T*. The term corresponding to this theorem is a function taking a type and two arrays over the given type as arguments and returning a proposition stating that the two arrays contain the same elements; that is,

$$\begin{aligned} term\_\_of(set\_\_eq\_\_) = {} & \lambda T.\lambda s.\lambda t. \\ & \forall i{:}\{1..s.1\}.\ \downarrow(\exists j{:}\{1..t.1\}.s.2(i)=t.2(j))\ in\ T\ \& \\ & \forall i{:}\{1..t.1\}.\ \downarrow(\exists j{:}\{1..s.1\}.t.2(i)=s.2(j)\ in\ T). \end{aligned}$$

The definition *set__eq* then defines the notation $s =_e t$ *over T* to be *term__of(set__eq__)(T)(s)(t)*. (Section 5.5 defines $\downarrow T$ for type *T*.)

The next three theorems in the library prove that *s=e t over T* is an equivalence relation for any small type *T*. The objects *set__eq__reflexive*, *set__eq__commutative*, and *set__eq__transitive* contain proofs that *s=e t* *over T* is reflexive, commutative and transitive, respectively; these theorems are useful in the definition of the type-valued function that constructs types of finite sets.

The theorem *finset__* defines a function which takes a small type as an argument and returns a type whose elements are the finite sets of elements from T, namely, the quotient type comprising arrays as the base type and *set__eq* as the quotient equivalence. Thus,

$$term\_\_of(finset\_\_) = \lambda T.(s,t){:}(n{:}int\#\{1..n\}\rightarrow T)//s =_e t \; over \; T.$$

6.2.2. Membership and Extensionality

In an array representation of finite sets the membership predicate for an element a and a set s essentially states that there is an index i into s such that $s[i]=a$. Our implementation uses this predicate to define set membership; however, the type-functional semantics of the Nuprl type theory and the semantics of the squash operator introduce some subtlety into the precise specification of membership. In Nuprl propositions are types, and predicates are therefore functions that return propositions (types). A polymorphic set membership predicate, then, takes a type, an element of that type, and a set and returns a type representing the proposition that the element is in the set. Since this predicate is a function, the type returned by the predicate must be equal for equal sets. Because of this constaint, the most obvious membership predicate,

$$\lambda T.\lambda a.\lambda s.\exists i{:}\{1..s.1\}.s.2(i)=a \; in \; T,$$

cannot be proven to be well-formed, because it is possible for arrays that are equal as sets to give rise to different membership types for the same element.

Less obviously, the membership predicate formed by squashing the body of the above lambda term also cannot be proven to be well-formed, again because different arrays that are equal as sets can give rise to squashed types that are not equal. Consider the term

$$M = \lambda T.\lambda a.\lambda s.\downarrow(\exists i{:}\{1..s.1\}.s.2(i)=a \text{ in } T),$$

and consider two arrays s and s' with $s =e\ s'$ *over* T for some small type T. For a in T, $M(T)(a)(s) = M(T)(a)(s')$ in U_1 exactly when

$$\downarrow(\exists i{:}\{1..s.1\}.s.2(i)=a \text{ in } T) = \downarrow(\exists i{:}\{1..s'.1\}.s'.2(i)=a \text{ in } T) \text{ in } U_1,$$

and this equality is provable in Nuprl exactly when one can prove that

$$(\exists i{:}\{1..s.1\}.s.2(i)=a \text{ in } T) \Leftrightarrow (\exists i{:}\{1..s'.1\}.s'.2(i)=a \text{ in } T).$$

However, the definition of $=_e$ itself contains squashed types, and the semantics of set types (which squashed types are defined in terms of) dictate that the predicate part of the set type is hidden when the set type is eliminated unless the goal is an equality type. This restriction forbids the proof of the bi-implication, since proving the bi-implication requires proving $(\exists i{:}\{1..s'.1\}.\ s'.2(i) = a \text{ in } T)$ under the assumption of $(\exists i{:}\{1..s.1\}.\ s.2(i) = a \text{ in } T)$ and using the squashed parts of the definition of $=_e$. However, the necessary hypotheses that the definition of $=_e$ contains will remain hidden and therefore inaccessible, and the proof cannot proceed.

The somewhat unintuitive solution to this problem involves squashing the body of term M again, yielding the term

$$M' = \lambda T.\lambda a.\lambda s.\downarrow\downarrow(\exists i{:}\{1..s.1\}.s.2(i)=a \text{ in } T).$$

This term can be proven well-formed, and the theorem object *in__* does so. The definition *in* then defines $a \in T\ s$ to be *term__of(in__)(T)(a)(s)*. Table 6.3 contains the definitions for membership, as well as the definitions for null and singleton sets, and table 6.4 lists the corresponding typing theorems.

Table 6.3. Nuprl Definitions for Membership, Null and Singleton.

Name	Display Form	Body
in	<a> ∈ <T> <s>	*term__of(in__)(<T>)(<a>)(<s>)*
null	∅ <T>	*term__of(null__)(<T>)*
singleton	{<a>}<T>	*term__of(singleton__)(<T>)(<a>)*

Table 6.4. Nuprl Typing Theorems for Membership, Null and Singleton.

Name	Statement
in__	$T{:}U_1 \rightarrow T \rightarrow U_1$
null__	$T{:}U_1 \rightarrow finset(T)$
singleton__	$T{:}U_1 \rightarrow T \rightarrow finset(T)$

Having defined a membership predicate, it now becomes necessary to prove that it meets the semantic specification of membership in traditional set theory; that is, for the above definition of set membership to be valid, we must be able to prove the extensionality axiom. The theorem *extensionality* contains a proof of the extensionality axiom. The statement of the theorem is slightly different that the usual statement of the extensionality axiom; our version reads

$\forall T{:}U_1.\ \forall s,t{:}finset(T).\ \downarrow(\forall a{:}T.\ a \in_T s \Leftrightarrow a \in_T t) \Leftrightarrow s = t \text{ in } finset(T)$,

whereas the usual version is not "squashed" and contains only the "only if" portion of the bi-implication. We elected to state the theorem in this form because there is some work involved proving the "if" part of the bi-implication in Nuprl (since substitution of equals for equals is not a trivial process in Nuprl), and since for any type T one can prove that $T \Rightarrow (\downarrow T)$, the statement of the "only if" part of the theorem is as strong as the classical statement of extensionality. The introduction of the squash operator on the left side of the bi-implication is necessitated by our proof methodology for this theorem; had we adopted a different proof style it would not have been necessary. Figure 6.1 lists the extensionality theorem, as well as the

extensionality

$\forall T{:}U_1.\forall s,t{:}\mathrm{finset}(T).\ \downarrow(\forall a{:}T.\ a \in_T s \Leftrightarrow a \in_T t) \Leftrightarrow s = t \text{ in finset}(T)$

null_axiom

$\forall T{:}U_1.\forall a{:}T.\ \neg(a \in_T \varnothing_T)$

singleton_axiom_1

$\forall T{:}U_1.\forall a,b{:}T.\ a = b \text{ in } T \Rightarrow a \in_T \{b\}_T$

singleton_axiom_2

$\forall T{:}U_1.\forall a,b{:}T.\ a \in_T \{b\}_T \Rightarrow a = b \text{ in } T$

Figure 6.1. Axiomatizations of Membership, the Null Set and Singleton Sets. The Nuprl theorem names are listed above each theorem.

theorems corresponding to the axiomatization of the null set and singleton sets, which are described in the next section.

6.2.3. The Null Set and Singleton Sets

This section describes the basic set constructors implemented in Nuprl. Traditional Zermelo-Fraenkel set theory axioms usually specify the existence of a null set and a pairing constructor that forms a set from two elements. The implementation described here defines and axiomatizes a null set, but instead of defining pairs it instead defines singleton sets. We choose to do this for reasons of efficiency; singleton sets appear frequently in the description of subsequent set-theoretic functions and in the implementation of CCS that follows, and having singleton sets implemented as efficiently as possible is therefore highly desirable.

A standard array implementation of set theory would (if empty arrays are possible) have the null set be an empty array and arrays containing one element represent "canonical" singleton sets. Our implementation follows this convention: for an arbitrary small type T, the Nuprl array $<0, \lambda x.x>$ denotes the null set, and for an arbitrary element a of T $<1, \lambda x.a>$ denotes the singleton set containing a. The theorem *null__* proves the well-formedness of this rendering of the null set, while the definition *null* defines the notation $\varnothing_T$ to be *term__of(null__)(T)*. The theorem *singleton__* proves the well-formedness of the above definition of singleton sets, while the corresponding definition *singleton* defines $\{a\}_T$ to be

term__of(singleton__) (T)(a).

The proofs of the appropriate axiomatic descriptions of these sets are straightforward; the theorem statements are listed in figure 6.1. The theorem *null_axiom* states that for any small type T $\varnothing_T$ has no elements. Theorem *singleton_axiom_1* states that if two elements in an arbitrary small type T are equal then the one element is a member of the singleton set containing the other. Theorem *singleton_axiom_2* states the converse, that if an element belongs to a singleton set containing another element then the two elements are equal.

6.2.4. Union, Family Union, Comprehension and Collection

The expressiveness of the ST model results directly from the presence of set-theoretic operations like union and collection that allow the more complicated program CCS constructors to be modeled. Accordingly, the implementation of finite sets defines and axiomatizes binary union and family union operators and collection and comprehension schemes.

Several notes about these operations are in order. We define both binary and family union, even though the first seems redundant in light of the second, because binary union turns out to be extremely useful in defining and axiomatizing both family union and comprehension. In another departure from standard presentations of set theory, collection and comprehension are represented as functions; collection maps a function and a set to a set, while comprehension maps a predicate and a set to a set. Representing these schemes in this fashion makes them easier to use later in our development of CCS. Finally, the axiomatizations of these operators occur in two parts, one of which corresponds to an "intro" rule and one of which corresponds to an "elim" rule. That is, one theorem states the conditions under which an element may be judged to be in a set constructed by one of the operators, while another theorem states what can be deduced about an element that belongs to a set built with one of these operators. This organizing principle enables us to write tactics that implement these axioms as proof rules. Table 6.5 lists the definitions of these operations, while table 6.6 lists the corresponding typing theorems.

Table 6.5. Nuprl Definitions for Union, Family Union, Comprehension and Collection.

Name	Display Form	Body
union	*<s> ∪ <T> <s>*	*term_of(union_)(<T>)(<s>)(<t>)*
set_union	*∪ <T> <s>*	*term_of(set_union_)(<T>)(<s>)*
comprehension	*{x∈ <s>| <P>& <p>} <T>*	*term_of(comprehension_)(<T>)(<P>)(<p>)(<s>)*
collection	*map <f> (<T> : <U>) on <s>*	*term_of(collection_)(<T>)(<U>)(<f>)(<s>)*

Table 6.6. Nuprl Typing Theorems for Union, Family Union, Comprehension and Collection.

Name	Statement
union__	$T{:}U_1 \rightarrow finset(T) \rightarrow finset(T) \rightarrow finset(T)$
set__union__	$T{:}U_1 \rightarrow finset(finset(T)) \rightarrow finset(T)$
comprehension__	$T{:}U_1 \rightarrow P{:}(T \rightarrow U_1) \rightarrow p{:}(P \text{ is decidable on } T) \rightarrow finset(T) \rightarrow finset(T)$
collection__	$T{:}U_1 \rightarrow U{:}U_1 \rightarrow (T \rightarrow U) \rightarrow finset(T) \rightarrow finset(U)$

6.2.4.1. Union

An array implementation of finite sets gives rise to a very efficient representation of the union of two sets. If the set equality has been defined appropriately (so that duplicate elements are ignored) then the concatenation of the two arrays has the properties of the union of the two corresponding sets. Since the concatenation of two arrays is obviously an array, the definition is valid, and the resulting union operator works in constant time. The Nuprl definition of binary union uses precisely this algorithm to compute the union of two sets. Defining concatenation of arrays in our framework involves some subtlety because array bounds can be negative; one must do more that just add together the bounds in order to compute the bound for the concatenation of the two arrays. The implementation circumvents this obstacle by defining a function, *nat*, over the integers that is written $\uparrow i$ for integer i and that returns 0 for nonpositive integers and the value of the input for positive integers. Given this definition, the term

$$\lambda s,t.<\uparrow s.1+\uparrow t.1,\lambda i.less(i;\uparrow s.1;s.2(i);t.2(i-\uparrow s.1))>$$

computes the concatenation of two arrays. The array bound of the result is the sum of the "naturalized" array bounds of the input arrays, and the array function first checks its argument to see whether it is in the range corresponding to the first array ($\{1..\uparrow s.1\}$) or the second array ($\{(1+\uparrow s.1)..(\uparrow s.1+\uparrow t.1)\}$) and returns the appropriate array element.

The Nuprl theorem *union__* proves the well-formedness of the aforementioned term. Unlike previous proofs of well-formedness, however, the proof that the term is functional over *finset(T)* for small types T required a substantial amount of effort. The difficulty of this proof stems from the complexity of the equality relation defined for *finset(T)* and from the relative ignorance that the Nuprl arithmetic decision procedure has for integer ordering. Using the extensionality theorem as a lemma, the proof of *union__* essentially splits into two symmetric pieces: under the assumptions that $<a_1,f_1>=_e<a_2,f_2>$ *over* T and $<a_3,f_3>=_e<a_4,f_4>$ *over* T, and given that

$$a\in_T<\uparrow a_1+\uparrow a_3,\lambda i.less(i;\uparrow a_1;f_1(i);f_3(i-\uparrow a_1))>,$$

show that

$$a\in_T<\uparrow a_2+\uparrow a_4,\lambda i.less(i;\uparrow a_2;f_2(i);f_4(i-\uparrow a_2))>,$$

and vice versa. The proof of one of these branches is rather involved, because lemmas like *monotonicity5* (if $i<j$ and $k<l$ then $i+k<j+l$), *lle__to__l* (if $i<j\leq k$ then $i<k$) and *natset1* ($\forall p:\{1..\uparrow i+\uparrow j\}.\ p<\uparrow i+1\Rightarrow 0<p\leq i$) had to be proven and invoked as lemmas. The final proof refers to

approximately 20 lemmas like this. The definition *union* defines the notation $s \cup_T t$.

The Nuprl theorems *union__axiom__1* and *union__axiom__2* are listed in figure 6.2; they prove that $\cup_T$ satisfies the properties of binary union.

union__axiom__1

$\forall T{:}U_1.\forall s,t{:}\mathbf{finset}(T).\forall a{:}T.\ a \in T\ s \cup T\ t \Rightarrow \downarrow (a \in T\ s \vee a \in T\ t)$

union__axiom__2

$\forall T{:}U_1.\forall s,t{:}\mathbf{finset}(T).\forall a{:}T.\ (a \in T\ s \vee a \in T\ t) \Rightarrow a \in T\ s \cup T\ t$

Figure 6.2. The Axiomatization of Union.

Theorem *union__axiom__1*, the "elim" axiom for union, states that if a is in $s \cup_T t$ then either a is in s or a is in t, while theorem *union__axiom__2*, the "intro" axiom, states that if a is in either s or t then it is in $s \cup_T t$.

6.2.4.2. Family Union

A binary union operator is insufficiently powerful to define all the functions that we are interested in; it turns out that we need a union operator that, given a finite set of finite sets, returns the union of all the constituent sets. In an array implementation of sets one can imagine implementing this union of a family of sets by recursively constructing the binary union of a constituent set with the family union of the family without the constituent set. The Nuprl development of this union operator follows this strategy by using the inductive definition form that the logic provides for integers. The term

$$\lambda s.ind(s.1\ ;\ u,v.\varnothing_T\ ;\ \varnothing_T\ ;\ u,v.s.2(u) \cup_T v)$$

takes an array corresponding to the family of finite sets as its argument and inductively analyzes the array bound of the argument. If the bound is negative or zero (in which case the family is empty) the term evaluates to the null set; if the bound is positive, the term evaluates to the binary union of the last element in argument array and the recursive evaluation of the term with the array bound decreased by one.

Theorem *set_union_* contains a proof of the well-formedness of this term with respect to elements in *finset(finset(T))*. Proving that the term is functional over finite sets of *finset(T)* requires some effort in that a double integer induction is necessary. The definition *set_union* defines the notation $\cup_T(S)$ for arbitrary small types T and families S of finite sets of T.

Proving the correctness of the development of family union follows more or less directly from the correctness of the implementation of binary union; figure 6.3 lists the associated Nuprl theorems. Theorem *set_union_axiom_1*, the "intro" rule for family union, states that if t is a constituent set in a family of sets s and a is in t then a belongs to the family union of s, while *set_union_axiom_2*, the "elim" rule, states the converse, with the computational content of the consequent necessarily squashed away.

set_union_axiom_1

$\forall T{:}U_1.\forall s{:}\mathbf{finset(finset(T))}.\forall a{:}T.$

$(\exists t{:}\mathbf{finset(T)}.\ t \in \mathbf{finset(T)}\ s\ \&\ a \in T\ t) \Rightarrow a \in T \cup T\ s$

set_union_axiom_2

$\forall T{:}U_1.\forall s{:}\mathbf{finset(finset(T))}.\forall a{:}T.$

$a \in T \cup T\ s \Rightarrow \downarrow(\exists t{:}\mathbf{finset(T)}.\ t \in \mathbf{finset(T)}\ s\ \&\ a \in T\ t)$

Figure 6.3. The Axiomatization of Family Union.

6.2.4.3. Comprehension

Given a predicate *P* on the elements of an array, one can derive a set-theoretic comprehension scheme informally in the following fashion. For an array *s*, iteratively test whether each element of *s* satisfies *P*. If *P* is true of an element, add it to the set being constructed; otherwise, leave it out of the set.

The Nuprl development of a comprehension operator follows this algorithm, but it differs in one major aspect. In Nuprl functions in the standard function space types must be total. However, for general predicates (where, in Nuprl, a predicate on a type *T* is a function from *T* to the collection of small types, U_1) it is not generally decidable whether a predicate holds of a particular element in the type in question. The previously described comprehension scheme, then, is well-formed in Nuprl only when the predicate is decidable. Accordingly, the Nuprl definition of comprehension allows only decidable predicates, which is to say only predicates for which it is provable that $\forall a{:}T.\ P(a) \vee \neg P(a)$. The definition object *decidable* defines the notation *P is decidable on T* for arbitrary predicates P and type T to be exactly this; hence, the type of the Nuprl term corresponding to set comprehension is $T{:}U_1 \rightarrow P{:}(T \rightarrow U_1) \rightarrow (P \textit{ is decidable on } T) \rightarrow \mathit{finset}(T) \rightarrow \mathit{finset}(T)$, where the third argument supplied to the form is a proof that P is decidable on type T and, given the constructive nature of the Nuprl proof theory, constitutes a decision procedure for P. (Alternatively, we could have defined the type of decidable predicates on a type T to be the functions of type $T \rightarrow \{0,1\}$ and used the decidability of integer equality to test for the truth or falsity of a particular predicate on a particular element. We opted for the former approach because the necessity of decidability is made more perspicuous.)

The term

$$\lambda P.\lambda p.\lambda s.$$

$$\mathit{ind}(s.1;\ u,v.\varnothing_T;\ \varnothing_T;\ u,v.\mathit{decide}(p(s.2(u));w.\{s.2(u)\}_T \cup_T v;x.v))$$

implements the algorithm described above for set comprehension. Given a predicate P, a decision procedure p for P and an array s, the body of the term inductively analyzes the array bound of s using the recursion scheme for integers in Nuprl. If the bound is less than or equal to zero, the array represents the null set, and the null set is therefore returned. If the bound is greater than zero, the last element in the array is tested using p. If the result is an *inl* term then the predicate P must be true of the element (since $p(a)$ for element a has type $P(a) \vee \neg P(a)$) and the array returned is the union of the singleton set containing the element with the set constructed recursively out of the rest of the array; otherwise, P must be false for the element and the array returned is the recursive construction of the rest of the set.

Theorem *comprehension_* proves the well-formedness of the above term. The proof had essentially the same complexity as the proof of the well-formedness of family union, which is to say that a double induction was necessary, as was a substantial amount of term-rewriting. The definition *comprehension* defines the notation $\{x \in s | P \ \& \ p\}_T$ for a finite set s, a predicate P, ad decision procedure p for P and a type T.

Theorems *comprehension_axiom_1* and *comprehension_axiom_2* embody the "intro" and "elim" forms, respectively, of the comprehension scheme and as such embody a proof of the correctness of our implementation; they are listed in figure 6.4. The first of these theorems

comprehension_axiom_1

$\forall T{:}U_1.\forall P{:}(T \to U_1).\forall p{:}(P \text{ is decidable on } T).\forall s{:}\text{finset}(T).\forall a{:}T.$

$a \in_T s \ \& \ P(a) \Rightarrow a \in_T \{x \in s|P\&p\}_T$

comprehension_axiom_2

$\forall T,U{:}U_1.\forall f{:}T \to U.\forall s{:}\text{finset}(T).\forall b{:}U.$

$b \in_U \text{map } f(T{:}U) \text{ on } s \Rightarrow \downarrow \exists a{:}T.f(a) = b \text{ in } U \ \& \ a \in_T s$

Figure 6.4. The Axiomatization of Comprehension.

states that for a given (decidable) predicate, set and element, if the predicate holds of the element then the element is in the set formed by comprehension from the given set and the predicate. The proof proceeds in a straightforward fashion by induction on the array bound of s. The second theorem proves that for a given (decidable) predicate, set and element, if the element is in the set formed by comprehension from the given set and the predicate then the element is in the given set and the predicate holds of it. Again, the proof proceeds by induction on the array bound of *s*.

6.2.4.4. Collection

An informal array implementation of finite sets would provide a collection scheme by iteratively applying the collection function to each element in the array. The higher-order programming aspect of the Nuprl system, however, allows a much simpler definition of collection based on function composition. Given a function *f* mapping small type *T* to small type *U* and a Nuprl array *s* of elements in type *T*, the collection of *f* over *s* can be represented as the array whose bound is the bound of *s* and whose enumeration function is the composition of *f* and the enumeration function of *s*. The term $\lambda f.\lambda s.<s.1,\lambda x.f(s.2(x))>$ effects this composition. Theorem *collection_* proves the well-formedness of this scheme, and the definition *collection* defines the notation *map f(T:U) on s* for function *f* of type $T \rightarrow U$ and *s* of type *finset(T)*. The correctness of this implementation follows from theorems *collection_axiom_1* and *collection_axiom_2* of figure 6.5, which also form the "intro" and "elim" rules, respectively, for this collection scheme. The statement of the first of these theorems says that if an element *a* is in a set *s* then *f(a)* is in *map f(T:U) on s*, while the statement of the second says that if an element belongs to *map f(T:U) on s* then an inverse image of the element with respect to *f* belongs to *s*.

collection_axiom_1
$\forall T,U:U_1.\forall f:T\rightarrow U.\forall s:\mathbf{finset}(T).\forall a:T.$
$a \in T\ s \Rightarrow f(a) \in U$ **map** $f(T:U)$ **on** s

collection_axiom_2
$\forall T,U:U_1.\forall f:T\rightarrow U.\forall s:\mathbf{finset}(T).\forall b:U.$
$b \in U$ **map** $f(T:U)$ **on** $s \Rightarrow \downarrow(\exists a:T.f(a)=b$ **in** $U\ \&\ a \in T\ s)$

Figure 6.5. The Axiomatization of Collection.

6.2.5. ST

This section concludes with a description of the type *ST*, the type corresponding to the finitary version of the process model developed in chpater 5. The type definition uses the recursive type constructor available in the Nuprl theory in conjunction with the finite set theory presented previously to implement in a rather direct fashion the recursively defined *ST* model.

The first task in deriving *ST* involves defining the actions a process is allowed to perform. It is possible to abstract the definition of *ST* with respect to the type of actions allowable to processes; in this framework *ST* would be a function mapping small types corresponding to types of actions to types whose elements are the *ST*-structures comprising actions of the argument types. However, the implementation described here does not follow this course, primarily because the algebraic formalisms we are most interested in (CCS) are only interesting in the context of actions for which, for example, equality is decidable. Therefore, we elect to model actions as integers, and accordingly, the definition *action* defines the notation *Action* to correspond to the Nuprl type *int*.

Given this definition of *Action*, the theorem *st__* addresses the well-formedness of the term defining *ST*. The term

rec(st.finset(Action #st))

defines a type consisting of finite sets of pairs of actions and elements of the type being defined, and its well-formedness follows in a straightforward fashion from the formation rules for recursive types. The Nuprl definition *st* defines the notation *ST* to be the term extracted from *st__*.

An unfortunate aspect of the recursive definition mechanism supplied by Nuprl concerns the fact that typing information involving the body of recursive definitions cannot be proven directly using the underscore convention. This can give rise to well-formedness subgoals that were previously caught. For example, using the set-theoretic operators defined previously as functions on *ST* requires "unrolling" *ST* by rewriting it as *finset(Action #ST)*; however, unrolling *ST* requires expanding the *term__of* extraction to yield *rec(st.finset(Action #st))* before the unrolling can take place. The unrolled type is then *finset(Action # rec(st.finset(Action #st)))*, and as the system has no indication that *rec(st.finset(Action #st))* has been proven well-formed already, a well-formedness subgoal appears. In order to circumvent this problem, the theorems *finset__to__ST* and *ST__to__finset* allow unrolling and rerolling to take place in a fashion that preserves type information. The first of these states that

$$\forall s{:}finset(Action\#ST).\ s\ in\ ST,$$

while the second states the converse, namely that

$$\forall s{:}ST.\ s\ in\ finset(Action\#ST).$$

Tactics that take advantage of these theorems help ameliorate the unwanted appearance of well-formed subgoals. Similar theorems are proven for other recursively defined operators.

6.3. Implementing CCS

This section describes the implementation of a model of CCS in terms of the type *ST*. In what follows the CCS action set will be taken to be the type of integers in Nuprl, with τ represented by the integer *0* and with complementation of actions denoted by integer negation. On occasion we shall refer to members of *ST* as sets of *branches*, where a branch has type *Action#ST*. When the first component of a branch is an action *a* we shall say that *a heads* the branch, and we shall ofter refer to the second component of a branch as a *continutation.*

We should note at the outset that we do not consider the CCS operation of relabelling because the example at the end of the library does not require it; however, in light of the other operators that we have developed, there is no inherent difficulty in developing a term that does relabelling. Tables 6.7 and 6.8 describe the definitions of the operators we do account for. The presentation of each operator has two parts--a proof of well-formedness and a proof that the defined term meets the semantic specification of the appropriate CCS operator.

Table 6.7. Nuprl Definitions of CCS Process Constructors.

Name	Display Form	Body
nil	*NIL*	*term__of(nil__)*
prepend	*<a><P>*	*term__of(prepend__)(<a>)(<P>)*
plus	*<P>+<Q>*	*term__of(plus__)(<P>)(<Q>)*
merge	*Σ<s>*	*term__of(merge__)(<s>)*
prune	*<P>-<a>*	*term__of(prune__)(<a>)(<P>)*
restrict	*<P>\<a>*	*term__of(restrict__)(<a>)(<P>)*
compostion	*<P>|<Q>*	*term__of(composition__)(<P>)(<Q>)*

Table 6.8. Nuprl Typing Theorems for CCS Process Constructors.

Name	Statement
nil__	*ST*
prepend__	*Action→ST→ST*
plus__	*ST→ST→ST*
merge__	*finset(ST)→ST*
prune__	*Action→ST→ST*
restriction__	*Action→ST→ST*
composition__	*ST→ST→ST*

Milner gives the semantics of the CCS term constructors in terms of derivation relations, where $P \rightarrow^{a} P'$ means intuitively that P performs an a action and evolves into P'. This semantics has a natural expression in the ST model, with set membership being the semantic criterion. That is, $P \rightarrow^{a} P'$ becomes $<a,P'> \in_{Action\#ST} P$. The Nuprl theorem *arrow__* proves that this translation is valid for ST, and the definition *arrow* defines the notation $P \rightarrow^{a} P'$ as the term extraction of *arrow__*.

6.3.1. Nil, Prepending, Plus and Merge

The four operators defined here have very straightforward descriptions as Nuprl/ST terms and thus are described together. The semantic theorems are contained in figure 6.6. The first of operators, NIL, corresponds to the null set in ST. Thus, the theorem *nil__* establishes the well-formedness of $\emptyset_{Action\#ST}$ with respect to ST, while the definition *nil* defines the notation NIL. The theorem *nil_semantics* verifies the semantic correctness of this implementation by proving that NIL admits no actions. The proof of this theorem follows immediately from *null_axiom*.

Prepending an action a to an existing term P (aP) can be represented by the singleton set $\{<a,P>\}_{Action\#ST}$. Accordingly, the theorem *prepend__* demonstrates that $\lambda a.\lambda P.\{<a,P>\}_{Action\#ST}$ is functional with respect to actions and elements of ST, and the definition *prepend* defines the notation aP. The theorem *prepend_semantics* establishes the semantic validity of this term by stating that aP may admit an a action and evolve to P.

nil__semantics
$\forall$P:ST. $\forall$a:Action. $\neg$(NIL$\rightarrow^{a}$P)

prepend__semantics
$\forall$P:ST. $\forall$a:Action. aP$\rightarrow^{a}$P

plus__semantics__1
$\forall$P,Q,P1:ST. $\forall$a:Action. P$\rightarrow^{a}$P1 $\Rightarrow$ P+Q$\rightarrow^{a}$P1

plus__semantics__2
$\forall$P,Q,Q1:ST. $\forall$a:Action. Q$\rightarrow^{a}$Q1 $\Rightarrow$ P+Q$\rightarrow^{a}$Q1

merge__semantics
$\forall$s:finset(ST). $\forall$P':ST. $\forall$a:Action.
($\exists$P:ST. P $\in_{ST}$ s & P$\rightarrow^{a}$P') $\Rightarrow$ Σs$\rightarrow^{a}$P'

Figure 6.6. Semantic Theorems for *NIL*, Prepending, Plus and Merge.

Summing two processes *P* and *Q* corresponds to taking the binary union of the *ST* representations of *P* and *Q*. The theorem *plus__* verifies that this operation is functional over *ST*, and definition *plus* defines the notation $P+Q$ for *P* and *Q* that are elements of *ST*. The semantic specification of + has two parts. Theorem *plus__semantics__1* establishes that if *P* admits an action and eveloves to *P'* then so does $P+Q$, while *plus__semantics__2* establishes the symmetrical result for *Q*. The proofs of these theorems follow immediately from the corresponding axioms of the union operator for sets.

Finally, summing together a finite number of processes may be described with the family union operator. Theorem *merge__* proves the well-formedness of $\lambda s.\cup_{Action\#ST}\ s$ over *finset(ST)*, and *merge* defines the notation Σs for aritrary finite sets s of elements of ST. The proof of theorem *merge__semantics* establishes the semantic validity of this definition by showing that for all finite sets of ST, if one of the elements of the set admits an action and evolves to a process P' then the merge of the set does as well.

6.3.2. Restriction

The representation of restriction is more complicated that the representations of the previous CCS operators because this operator has a recursively specified semantics; therefore, the corresponding Nuprl term must be defined recursively. The equational specification for restriction with respect to an action a and a CCS term $P = \Sigma a_i P_i$ is as follows; in what follows, $\Sigma a_i B_i\,(\phi)$ will refer to the merge of branches $a_i B_i$ satisfying ϕ.

$$P\backslash a = \Sigma a_i(P_i\backslash a)\ (a_i \neq a \vee a_i \neq \underline{a})$$

This recursive description of restriction has a straightforward encoding as a Nuprl term, as the remainder of this section will show.

The implementation of restriction is given in terms of a so-called "prune" operator that recursively prunes away the elements in a given ST whose action part equals a given action. Equationally, *prune* is defined as $prune(a,P) = \Sigma a_i prune(a,P_i)\ (a_i \neq a)$. In Nuprl this is best done with the comprehension scheme defined above, so the first step in defining a pruning operator involves defining and proving decidable the predicate described informally above. Theorem *prune_predicate_* proves that the term

$$\lambda a.\lambda x.(\neg x.1 = a \ in \ Action)$$

has type

$$Action \rightarrow (Action\#ST) \rightarrow U_1,$$

meaning that for a given action a this term applied to a yields a predicate on *Action#ST*. The definition *prune_predicate* defines the notation $\neg$ *action part*$=a$ to be the predicate just described, given an arbitrary action a. The theorem *prune_predicate_decidable* then establishes that $\forall a{:}Action.\neg$ *action part*$=a$ *is decidable over Action#ST*.

Given these definitions, the following Nuprl term computes *prune*.

$\lambda a.\lambda s.rec\_ind(s; prune,t.$

$\quad map\ \lambda b.<b.1,prune(b.2)>_{(Action\#ST:Action\#ST)}$

$\quad on\ \{x \in t \mid \neg action\ = a\ \&$

$\qquad term\_of(prune\_predicate\_decidable)(a)\}_{Action\#ST}.$

On action argument a and ST argument P this term applies the recursive invocation of the function being defined to the continuation of each branch in P with branches headed by a removed. Theorem *prune_* proves the well-formedness of this term, and definition *prune* defines the notation P-a for arbitrary ST P and action a. Theorem *prune_semantics* proves that if $P \rightarrow^a P'$ and $b \neq a$ then $P\text{-}b \rightarrow^a P'b$ ($\forall P, P_1{:}ST.\ \forall a,b{:}Action.\ a \neq b \Rightarrow P \rightarrow^a P_1 \Rightarrow P\text{-}b \rightarrow^a P_1\text{-}b$).

Theorem *restriction_* proves that $\lambda a.\lambda s.(s\text{-}a)\text{-}(\text{-}a)$ is well-formed, and definition *restriction* defines the notation $P\backslash a$ for arbitrary ST P and action

a. The obvious semantic theorm, *restriction_semantics*, follows directly from *prune_semantics*.

6.3.3. Composition

The CCS composition operator has a straightforward equational description given by the expansion theorem, which states that if

$$P = \Sigma a_i P_i$$

and

$$Q = \Sigma b_i Q_i$$

then

$$P|Q = \Sigma a_i(P_i|Q) + \Sigma b_i(P|Q_i) + \Sigma\tau(P_i|Q_i)(a_i = \underline{b_i}).$$

However, giving an account of composition involves some complexity that belies this simple equational specification, complexity that stems from a certain inflexibility in the Nuprl recursive definition operator. The most natural definition of $P|Q$ for P and Q in ST involves an interleaved induction on the structure of P and Q; recursive definition in Nuprl, however, only allows the inductive analysis of one recursively defined object at once. Nonetheless, we can rewrite the equational specification of composition in terms of two mututally recursive functions, *rcomp* and *lcomp*, that are each recursive in only one of the two arguments to composition. Consider the following system of equations.

$lcomp(P) = \lambda Q.rcomp\,(P,Q)$

$rcomp(P,Q) = \quad \Sigma a_i lcomp(P_i)(Q) +$

$$\Sigma b_j rcomp(P,Q_j) +$$
$$\Sigma\tau lcomp(P_i)(Q_j)\,(a_i = \underline{b}_j)$$

That this a valid specification of composition follows by performing the substitution for *lcomp* specified by the first equation. It is also the case that *lcomp* is inductively defined only in terms of the first argument, P, to the composition operator, while *rcomp* is inductively defined only in terms of its second argument, Q. This specification therefore suggests a form for the Nuprl term, with two nested recursive forms implementing the mutally recursive definitions of *lcomp* and *rcomp* and the body of the inner recursive specification using the collection and comprehension operators defined previously to realize the specfication of *rcomp.*

The definition of composition requires two uses of the comprehension scheme with different predicates. One of the predicates is *prune_predicate* and therefore has already been defined as suitable for set comprehension; the second is the negation of *prune_predicate.* This predicate is defined by theorem *weed__*, which establishes that

$$\lambda a.\lambda b.a = b.1 \; in \; Action$$

has type

$$Action \rightarrow (Action\#ST) \rightarrow U_1,$$

and the definition *weed* defines the notation *action part* $=$ *a* for arbitrary actions *a* to be the application of this term to *a*. Theorem *weed__decidable* establishes the decidability of this predicate by appealing to the decidability of *prune_predicate.*

Given these definitions the term contained in figure 6.7 computes the composition of two elements of *ST*. Operationally, the composition term

```
λs. rec_ind (s; lcomp, s1.
    λt. rec_ind(t; rcomp, t1.
        map (λb. <b.1, lcomp(b.2)(t1) >) on s1 +
        map (λb. <b.1, rcomp(b.2) >) on t1 +
        Σ  map (λc.  map (λb. <0,lcomp(b.2)(c.2) >) on
                          {x  ∈  {x∈s1 | ¬action part = 0 & term_of
                                 (prune_predicate_decidable)(0)}
                              |  action part = -c.1 &
                                 term_of(weed_decidable)(-c.1)}
          on t1))
```

Figure 6.7. The Nuprl Term for Composition.

behaves in the following fashion. Given two arguments P and Q of type ST, the outer *rec_ind* form defines *lcomp* while the inner *rec_ind* form defines *rcomp*. Each of these recursive forms has its principal argument followed by a semicolon and a term that binds two names to a function representing the recursive invocation of the form being defined and the value of the principal argument, respectively. The body of the inner *rec_ind* has three terms connected by a CCS $+$; these three terms correspond to the three parts of the recursive definition of *rcomp* in the equational specification above. One sees that the first term implements $\Sigma a_i lcomp(P_i,Q)$ by noting that $\Sigma a_i lcomp(P_i,Q)$ actually specifies the application of *lcomp* to the continuation of each branch in P and to Q; in the set model, this represents a straightforward application of collection, with the collection function mapping a branch b to a branch consisting of the action part of b and the recursive application of *lcomp* to the continuation of b and to Q. The function

$$\lambda b.<b.1, lcomp(b.2)(t1)>$$

calculates this collection function because $t1$ is bound to Q. Thus,

$$map\ (\lambda b.<b.1, lcomp(b.2)(t1)>)\ on\ s1$$

describes $\Sigma a_i lcomp(P_i,Q)$. Similarly,

$$map\ (\lambda b.<b.1, rcomp(b.2)>)\ on\ t1$$

implements $\Sigma b_j rcomp(P,Q_j)$ as an instance of collection, with the collection function applying *rcomp* to P (implicitly, since it is evaluated in an environment that has P bound to $s1$) and to each continuation in Q. In a more roundabout fashion

$$
\begin{aligned}
&\Sigma map\ (\lambda c.map \quad (\lambda b.<0,lcomp(b.2)>)\ on \\
&\qquad\qquad \{x \in \{x \in s1 \mid \neg action\ part = 0\ \& \\
&\qquad\qquad\quad term\_of\ (prune\_predicate\_decidable)(0)\} \mid \\
&\quad action\ part = \text{-}c.1\ \& \\
&\quad term\_of(weed\_decidable)(\text{-}c.1)\}) \\
&\quad on\ t1
\end{aligned}
$$

corresponds to $\Sigma\tau lcomp(P_i,Q_j)\ (a_i = \underline{b}_j)$. Given P and Q of type ST, one may intuitively calculate $\Sigma\tau lcomp(P_i,Q_j)\ (a_i = \underline{b}_j)$ by, for each branch in Q, computing the set of branches headed by the inverse of the action heading the branch from Q and producing the ST whose branches are headed by τ and whose continuations are the results of the composition of the continuations of the branch from Q with each continutation in this set, and at the end of this process merging all the resulting elements of ST. The Nuprl term above formalizes this algorithm. On a branch c of $Q\,(t1)$,

(λc.map (λb. <0,lcomp(b.2)(c.2)>) on

{x ∈ {x ∈ s1 | ¬action part = 0 &

term__of (prune__predicate__decidable)(0)} |

action part = -c.1 & term__of(weed__decidable)(-c.1)})

takes a set that has all branches headed by actions that cannot synchronize with *c.1* (note that branches headed by τ--*0* in our representation--can never synchronize with anything and hence are always removed) and produces an ST whose branches are headed by τ and followed by elements of ST resulting from the composition of *c.2* with continations from the set. When mapped onto Q (*t1*), this function produces a set of elements of ST that, when merged together, produce the desired result.

Theorem *composition* proves that this term is functional on ST, and the definition *composition* defines the notation $P|Q$ for arbitrary STs P and Q. Establishing the semantic validity of this definition of relies on first proving that the expansion theorem holds of $P|Q$. The theorem *expansion__thm* contains a proof of this; its statement is listed in figure 6.8, along with the statments of the semantic theorems for composition.

6.3.4. The Binary Semaphore Example

This section concludes with a presentation of the demonstration in the library that the binary semaphore in [Milner 80] is correct. This demonstration uses functions defined in Nuprl for listing elements of ST to *expansion__thm*

$\forall$P,Q:ST.

P|Q = map (λb.<b.1, P|b.2>) on t1 +
map (λb.<b.1, b.2|Q>) on z +
Σ map (λc. map (λb.<0, b.2|c.2>) on
{x $\in$ {x$\in$P | $\neg$action part = 0 & term_of
(prune_predicate_decidable)(0)} | action part = -c.1 &
term_of(weed_decidable)(-c.1)} on Q in ST

composition__semantics__1

$\forall$P,Q,Q':ST. $\forall$a:Action. $Q \rightarrow^{a} Q' \Rightarrow P|Q \rightarrow^{a} P|Q'$

composition__semantics__2

$\forall$P,Q,P':ST. $\forall$a:Action. $P \rightarrow^{a} P' \Rightarrow P|Q \rightarrow^{a} P'|Q$

composition__semantics__3

$\forall$P,Q,P',Q':ST. $\forall$a:{x:Action|x$\neq$0}. $P \rightarrow^{a} P'$ & $Q \rightarrow^{-a} Q' \Rightarrow P|Q \rightarrow^{0} P'|Q$

Figure 6.8. Theorems for Composition.

show that mutual exclusion is enforced in the example. Primary among these is the function *list__traces*, contained in the eval object *test*, that lists the traces of an element of *ST'*. This function consists of the composition of two functions: *traces*, which generates a *finset* containing lists of actions that are the traces of the argument, and *list__set*, which produces a list whose elements are the elements of the set argument. Function *traces* is bound to the following term.

λ*s.rec__ind(s; lister, t. int__eq(t.1;0;*

$\{nil\}_{int\ list};$

$\cup_{int\ list}$(*map* λb.(*map* λl.((*b*.1).*l*) *on lister*(*b*.2))
on t)))

Intuitively, on an input of type *ST traces* recursively tests whether the input corresponds to the CCS *NIL* term (it does if the array bound--*t.1* in the above term--is equal to *0*). If the term is *NIL* then the form returns the set consisting of the empty trace; otherwise, *traces* computes a set of traces for each branch in the input *ST* and unions these sets together. Doing this involves using the collection scheme twice to construct a set of set of traces. The collection function of the outer invocation of collection takes a branch as input and constructs a set of traces by recursively applying the *rec_ind* term (in the form of *lister*) to the continuation of the branch (*b.2*) and then prepending the action heading the branch (*b.1*) to each element of this set ((*b.1*).*l* does this).

The term for *list_set* is λ*s.ind(s.1;u,v.nil;nil;u,v.s.2(u).v)*. On array input *s*, this term inductively analyzes the array bound (*s.1*). If the bound is negative then the array (and corresponding set) is empty and the empty list (*nil*) is returned (*u,v.nil*); similarly, if the array bound is zero then *nil* is returned (*nil*). If the bound is positive then the list returned is the last element in the array consed with the recursively constructed list for the array minus its last element (*u,v.s.2(u).v*). Term *list_traces*, then, is defined as λ*s.list_set(traces(s))*.

Eval object *mutex_defs* contains definitions for actions *a*, *b*, *c*, *d*, *P* and *V* and for CCS terms p_1 (*PabVNIL*), p_2 (*PcdVNIL*), and *sem* (*-P-V-P-VNIL*). Eval object *mutex_eval* then defines *l* as *list_traces* $(p_1|sem|p_2)\backslash P\backslash V$, and after 10 minutes of computation, the result (*l* has the value

$(0.a.b.0.0.c.d.0).(0.c.d.0.0.a.b.0))$ establishes that the only valid traces do not allow ab and cd to be interleaved. It is also possible possible to prove that $(p_1|sem|p_2)\backslash P\backslash V = 0ab00cd0 + 0cd00ab0$ *in* ST using Nuprl, although the proof would be very tedious, owing to the very small steps of inference allowed by the equational rules the CCS operators satisfy. It is worth noting, however, that equality in ST is decidable; therefore, the best way to prove theorems like this would be to prove this decidability theorem in Nuprl and then invoke the corresponding decision procedure.

6.4. On Using Nuprl

Using a system like Nuprl to develop a particular theory invariably makes one aware of certain things that the system does well and certain things that the system does poorly. While this section will not be a complete account of the experiences that accompanied the implementation of this theory of concurrency in Nuprl, several aspects of the system deserve comment.

One of the stongest reasons for using Nuprl involves the facilities the system provides for hiding and encapsulating information, both in the type theory, where set types allow the suppression of computational content of the set predicate, and in the proof theory, where definitions and tactics may be defined that create new, higher-level notations from lower-level ones. The set types prove to be extremely important in the implementation just described, for these types allow the definition of the squash operator, which removes enough computational content to allow certain terms to be functional over given types. As for definitions and tactics, while notation is unimportant in theoretical investigations into the foundations of mathematics, it is vitally important if one is to use the formal systems resulting from these investigations to develop real theories. Having used the system exhaustively, we conclude that any implementation of a formal system that is to be of practical use must have such facilities.

Another benefit to be gained from using Nuprl would also hold of any formal system, and that is the consistency checks the logic places on developments of theories inside the logic. Our first account of union, for example, was incorrect in that it neglected to take account of negative array bounds, and this was brought home to us when we were unable to prove that the union axioms were satisfied by our implementation of union. Although this example is relatively minor, when one handles very complicated objects this aspect of the theory becomes very important; a compelling case of this has been presented by Howe [Howe 87].

One interesting aspect of doing mathematics in an automated formal system is that efficiency of defined functions becomes important. Thinking computationally in a proof-theoretic setting proved to be more difficult that we first imagined, and as a result several of our implemented functions, although correct, had to be re-implemented in order to behave feasibly. In particular, the first derivation of *prune* had the recursive call taking place before the pruning took place; the resulting function was correct, but we were unable to carry out the semaphore example because the function took too long to compute.

Certain aspects of the theory and implementation do need to be re-examined. On the the theoretical side, the well-formedness subgoals that crop up after most invocations of proof rules are very distracting; in general we would advocate separating proofs of well-formedness from proofs of theorems, when possible. On the system side, a truly interactive ML interface would immensely ease the task of writing tactics. Also, some form of incremental proof updating would greatly increase user efficiency. As the system stands, any change in a proof rule causes proofs of children to the altered node to be completely lost. The problem is a difficult one to address, although the work of Griffin [Griffin 87] appears to hold promise. A reworking of the definition mechanism would also be beneficial. Currently, any substitution into a definition cause the display form of the definition to be lost. This can be disastrous when one has very complex objects, because the structure of the object is lost and can be extremely difficult to recover at times.

Chapter 7

Parallel Evaluation in Nuprl

The previous chapters of this thesis have considered Nuprl as a metatheory for developing models for reasoning about concurrency. Models have been described as types, and the Nuprl logic has been used to manipulate objects in these types. This chapter changes tack and instead examines incorporating concurrency into the Nuprl theory itself, thereby allowing the use of the theory to reason implicitly about parallelism. The appeal of this idea lies in the proofs-as-programs paradigm that underlies the interpretation of Nuprl as a programming language. Introducing a notion of parallel evaluation would allow programs that execute in parallel to be extracted from the proofs of theorems, programs whose correctness would be guaranteed by the correctness of the extraction process and the soundness of the Nuprl logic. It should be noted that we consider the Nuprl theory without recursive types in this chapter.

7.1. Evaluation in Nuprl

The evaluation strategy employed by Nuprl dictates that terms be evaluated in a sequential, outermost-redex-first fashion. The outermost redex is reduced according to a set of rewrite rules until the outermost term constructor is irreducible (canonical). The evaluation procedure then terminates, and the value of the term being evaluated is the result of the evaluation procedure.

The Nuprl term language is functional, and the size of the body of literature for evaluating functional language programs in parallel would suggest that "parallellizing" the evaluation procedure in Nuprl would introduce the desired style of parallelism into Nuprl term evaluation. Unfortunately, the head-first reduction strategy, coupled with the definition of canonical form, sharply limits the amount of computation that may be done concurrently. Accordingly, this section has two parts. The first part presents a new evaluation scheme that preserves the semantics of the Nuprl theory while offering more opportunities for parallelism than the current scheme. The second part examines one method based on strictness analysis [Mycroft 80] for introducing parallelism into the new evaluation scheme.

7.2. Evaluating Eagerly

As was said above, the current Nuprl evaluation scheme does not admit much parallel evaluation, primarily because the scheme is designed so that a minimum amount of computation is done in order for a normal form to be reached. The Nuprl evaluation scheme is described in figure 7.1. A term is

```
let eval(t:term) =
    case t is
    canonical:   return t
    noncanonical, with principal arguments p_i:
            return eval (contract t with eval(p_i))
end
```

Figure 7.1. The Nuprl Evaluation Procedure.
The phrase *contract... with...* refers to the redex contractions defined in chapter 3.

canonical if its outermost term constructor is canonical; therefore, $<0, 1+1>$ is canonical. If a term is canonical, *eval* returns the term. If a term is not canonical, *eval* evaluates the principal arguments to the form, computes the contractum associated with the noncanonical form (as described in chapter 3) and evaluates the result.

One means of introducing concurrency into Nuprl evaluation involves redefining Nuprl evaluation so that more computation is performed. This section presents such an evaluation scheme and demonstrates that the semantics of the Nuprl theory are respected by the scheme in the sense that computation of typed terms still terminates.

7.2.1. The Evaluation Scheme

The presentation of the new evaluation scheme has several parts. We first introduce a relation on terms that the new evaluation scheme will compute and show that the new evaluation scheme *respects* the Nuprl type system in the sense of [Allen 87b]: if two terms exhibit the relation then they are either untyped or have the same type and are equal in the type. This relation will then be shown to induce a new notion of canonical term, and an algorithm for computing the canonical representative of a term is then given and proven correct.

In order to make the presentation that follows clearer, we define the following terminology over Nuprl terms. A term constructor is a *base* term constructor if it contains no proper subterms. Examples include integer constants as well as the type expressions *int* and *void*. A term constructor is a *term synthesizer* if it has proper subterms and if instances of the term constructor are canonical. Examples of term synthesizers include the pairing operator ($<x,y>$) and lambda terms. A term whose outermost term constructor is a term synthesizer is a *synthesizer* term. A term constructor is a *term analyzer* if it has proper subterms and is not a term synthesizer. Examples include *spread* terms and function application. A term whose outermost constructor is a term analyzer is an *analyzer* term. A term that is base or a synthesizer term is called a *canonical* term. The *arguments* in a term constructor are the positions occupied by the largest proper subterms. We shall also write $s<t$ when the result of invoking the current Nuprl

evaluation procedure on term t terminates with result s. We shall also say that t evaluates to s when $s<t$.

We now make the following definitions.

Definition 7.1: A *context* is a term with metavariables replacing some subterms. For clarity, metavariables will be doubly underscored capital letters. Examples of contexts include $x{:}A\,\#\underline{\underline{T}}$, $spread(x;u,v.\underline{\underline{T}})$ and $x{:}\underline{\underline{T}}\rightarrow B$. A term *occurs in a context* if it is a subterm of another term and the result of replacing the first term with a metavariable is a context.

Definition 7.2: A *binding context* is a context in which open variables in terms that are substituted for metavariables may become bound. In the previous examples $x{:}A\,\#\underline{\underline{T}}$, and $spread(x;u,v.\underline{\underline{T}})$ are binding contexts because terms substituted for $\underline{\underline{T}}$ will have open instances of x (in the first example) and u and v (in the second example) bound.

Definition 7.3: A subterm of a synthesizer term is *reducible* if it does not occur in a binding context.

Table 7.1 lists canonical term constructors, with underscores marking positions occupied by reducible subterms. Table 7.2 lists term analyzers.

For convenience, we shall on occasion write terms in prefix form, with the term constructor appearing first followed by the reducible arguments in the case of term synthesizers (principal arguments in the case of term analyzers), the binding variables and the nonreducible arguments. Thus

Table 7.1. Canonical Terms.
x, y represent variables.
n represents integer constants.
a represents atom constants.
Boldface (**a**, **b**, **A**, **B**) represent term metavariables;
underlined metavariables denote reducible positions.

		Base Terms		
x	n	a	*axiom*	*nil*
void	*int*	*atom*	U_i	
		Term Synthesizers		
$\underline{\mathbf{a}}.\underline{\mathbf{b}}$	$inl(\underline{\mathbf{a}})$	$inr(\underline{\mathbf{a}})$	$<\underline{\mathbf{a}},\underline{\mathbf{b}}>$	$\lambda x.\mathbf{b}$
$\underline{\mathbf{a}}<\underline{\mathbf{b}}$	$\underline{\mathbf{A}}\ list$	$\underline{\mathbf{A}}\mid\underline{\mathbf{B}}$	$\underline{\mathbf{A}}\#\mathbf{B}$	$x{:}\underline{\mathbf{A}}\#\mathbf{B}$
$\underline{\mathbf{A}}\to\mathbf{B}$	$x{:}\underline{\mathbf{A}}\to\mathbf{B}$	$\underline{\mathbf{A}}//\mathbf{B}$	$(x,y){:}\underline{\mathbf{A}}//\mathbf{B}$	$\{\underline{\mathbf{A}}\mid\mathbf{B}\}$
$\{x{:}\underline{\mathbf{A}}\mid\mathbf{B}\}$	$\underline{\mathbf{a}}=\underline{\mathbf{b}}\ in\ \underline{\mathbf{A}}$			

Table 7.2. Term Analyzers.
u,v,x,y,z represent variables.
Boldface (**a**, **b**, **c**, **d**) represent term metavariables; underlined metavariables denote principal arguments.

$\underline{\mathbf{a}}+\underline{\mathbf{b}}$	$\underline{\mathbf{a}}-\underline{\mathbf{b}}$	$\underline{\mathbf{a}}*\underline{\mathbf{b}}$
$\underline{\mathbf{a}}/\underline{\mathbf{b}}$	$\underline{\mathbf{a}}\ mod\ \underline{\mathbf{b}}$	$less(\underline{\mathbf{a}};\underline{\mathbf{b}};\mathbf{c};\mathbf{d})$
$int\_eq(\underline{\mathbf{a}};\underline{\mathbf{b}};\mathbf{c};\mathbf{d})$	$ind(\underline{\mathbf{a}};u,v.\mathbf{b};\mathbf{c};x,y.\mathbf{d})$	$atom\_eq(\underline{\mathbf{a}};\underline{\mathbf{b}};\mathbf{c};\mathbf{d})$
$list\_ind(\underline{\mathbf{a}};\mathbf{b};x,y,z.\mathbf{c})$	$decide(\underline{\mathbf{a}};x.\mathbf{b};y.\mathbf{c})$	$spread(\underline{\mathbf{a}};x,y.\mathbf{b})$
$\underline{\mathbf{f}}(\mathbf{a})$		

$x{:}A\#B$ will be written as $\#(A;x;B)$, $(x,y){:}A//B$ as $//(A;x,y;B)$, and $decide(t_1;u.t_2;v.t_3)$ as $decide(t_1;u,v;t_2,t_3)$. Generic terms have the form

$T(\underline{r};\underline{x};\underline{b})$, where T is a term constructor, $\underline{r}=r_1,\ldots,r_n$, $\underline{x}=x_1,\ldots,x_m$, and $\underline{b}=b_1,\ldots,b_l$.

The definition of the relation on terms follows.

Definition 7.4: Let s and t be Nuprl terms. Then $s \lll t$ is the strongest relation satisfying the following: there is a term synthesizer C such that

(i) t evaluates to $C(\underline{a};\underline{x};\underline{b})$ and s is $C(\underline{a'};\underline{x};\underline{b})$, and

(ii) for all i in the range $1 \leq i \leq n$, $a_i' \lll a_i$.

Intuitively, $s \lll t$ if s is "more computed" than t in the sense that s has the same outermost synthesizer as the result of evaluating t and that the reducible positions of s are computed further than the reducible positions of the result of evaluating t. It is worth noting that the definiton of $\lll$ implies that if $s \lll t$ and $t' < t$ then $s \lll t'$.

The following lemma will be necessary in the exposition that follows. The relation $=_\alpha$ on terms denotes syntactic equivalence up to α-conversion. A two-place relation, $R(x,y)$ is α-*functional* in argument x whenever $R(a,b)$ and $R(a,c)$ implies that $c =_\alpha d$.

Lemma 7.1: $\lll$ is α-functional in its first argument.

Proof: Suppose $s \lll t$ and $s' \lll t$; we must show that $s =_\alpha s'$. The proof follows by a trivial induction on the structure of s.

□

As a result of this lemma, we need not concern ourselves with the names of bound variables in what follows.

The rest of this section presents a proof that $\leqslant$ respects the Nuprl type system; that is, we shall prove that if $t \in T$ and $s \leqslant t$ then $s = t \in T$. Doing so requires a formal specification of the semantic relations $t \in T$ and $s = t \in T$ for Nuprl. The most complete account of the semantics of Nuprl occurs in Allen's thesis [Allen 87b], and this account will form the basis of the argument that follows. While it is beyond the scope of this chapter to include the entire semantic account of Nuprl, the semantics of the theory dictate that $t \in T$ exactly when $t = t \in T$, and $s = t \in T$ is defined only if T is a type and s and t are elements of the type T. Furthermore, T is a type exactly when T evaluates to a canonical type expression (that is, there exists a canonical term T' such that $T' < T$ and T' is a type expression), and s and t are equal elements of T if and only if they evaluate to equal canonical terms in S (that is, there exist terms s' and t' such that $s' < s$, $t' < t$ and $s' = t' \in T'$). Canonical terms and type expressions, and the equality relations relating them, are defined recursively in an intuitive way.

Lemma 7.2: If $t \in T$ and $s =_{\alpha} t$ then $s = t \in T$.

Proof: Follows from the capture-free nature of Nuprl substitution.

□

Lemma 7.3: The following hold.

(i) If $<a,b>\in\{x:A|B\}$ then there are types A', B' with $a\in A'$ and $b\in B'$.

(ii) If $inl(a)\in\{x:A|B\}$ then there is a type A' with $a\in A'$, and if $inr(b)\in\{x:A|B\}$ then there is a type B' with $b\in B'$.

(iii) If $a.b\in\{x:A|B\}$ then there is a type A' with $a.b\in A'$ *list*.

(iv) If T is a canonical type expression with $T\in\{x:A|B\}$ then there is a universe level k with $T\in U_k$.

The analogous results hold for quotient types.

Proof: By induction on the structure of A. By definition of membership in $\{x:A|B\}$ if $t\in\{x:A|B\}$ then $t\in A$. If A evaluates to a set or quotient type, the result follows by induction. Suppose then that A does not reduce to a set or quotient type. Each of the four cases above must be examined separately.

(i) According to the Nuprl semantics, the existence of the term synthesizer $<,>$ dictates that A reduce to $x:A'\#B'$ for some type A' and type family B'. The definition of membership for pair requires that $a\in A'$ and $b\in B'[a/x]$, so the result holds.

(ii) The term synthesizer *inl* requires that A reduce to $A'|B'$ for types A' and B'. As $inl(a)\in A'|B'$ if and only if $a\in A'$ the result holds. A symmetric argument establishes the result in the case of $inr(b)$.

(iii) The term synthesizer . requires that A reduce to A' *list* for some type A', and the result follows.

(iv) Since T is a canonical type expression A must reduce to U_k for some k, and the result follows.

□

Lemma 7.4: If t is a synthesizer term with $t \in T$ then there exist types $T_1,\ldots,T_n$ such that for each reducible subterm r_i in t, $r_i \in T_i$.

Proof: By induction on the structure of t. The argument for each case is similar, so we examine only one in detail.

<u>$t = \langle a,b \rangle$:</u> By definition of $\in$ there must be a T' such that $T' < T$ and $t \in T'$; it must also be the case that T' is either a product type, a set type or a quotient type. In the latter two cases lemma 7.3 gives the result. In the first case, T' is $x{:}A\#B$, where A is a type and B is a type family indexed by elements of A. Now $a \in A$ and $b \in B[a/x]$ and the result holds.

□

Theorem 7.1: If $t \in T$ and $s \ll t$ then $t = s \in T$.

Proof: Suppose $t \in T$ and $s \ll t$. From the semantic specification of Nuprl, there exist canonical terms t' and T' such that $t' < t$, $T' < T$, T' and T are equal types, and $t' = t \in T'$. It suffices, then, to show that $s = t' \in T'$. The proof proceeds by induction on the structure of t'.

<u>t' is base:</u> In this case, since t' has no reducible positions and no variable occurrences, lemma 7.1 implies that s and t' are the same term, and this fact implies that $s = t' \in T'$.

<u>t' is a synthesizer term:</u> In this case there exists a term synthesizer C such that t' is $C(\underline{a};x;\underline{b})$ with each a_i reducible. By lemma 7.4 there exist types T_i such that $a_i \in T_i$, and the definition of $\ll$ dictates that s is $C(\underline{a'};x;\underline{b})$ with each

$a'_i \leqslant a_i$. By induction it follows that $a_i = a'_i \in T_i$, and the definitions of equality on members allows us to conclude that $s = t' \in T'$.

□

7.2.2. Computing $\leqslant$

This section describes an algorithm, *reduce*, that reduces terms to a form that is in some sense "more reduced" than canonical form as described by Constable, et al. [Constable et al. 86] We then prove that *reduce* in fact computes $\leqslant$ in the sense that $reduce(t) \leqslant t$ for any term t.

To begin with, we first define the notion of *reduced* form as in table 7.1; the intent is that *reduce* will return a reduced form following this definition. For comparison, the canonical terms are listed also. Intuitively, canonical terms are reduced if the subterms that do not appear in a binding context are reduced. Accordingly, $<1,1+1>$ is not reduced because $1+1$ is not reduced and does not appear in a binding context, while $\lambda x.1+1$ is reduced because $1+1$ appears inside a binding context, namely λx.

The reduction procedure appears in figure 7.2. Intuitively, this procedure works in the following fashion. On term t, if t is a base term then t is the result of the reduction. If t is a synthesizer term, then t is returned with its reducible positions reduced. If t has a term analyzer as its outermost constructor then its principal arguments are evaluated, the contractum computed, and the result of the contractum reduced.

Table 7.3. Canonical and Reduced Forms.
a, b, A, B represent terms.
x, y are variables.

Canonical forms	Reduced forms	
x	x	
n	n	
a	a	
axiom	*axiom*	
nil	*nil*	
$a.b$	$a.b$	a, b reduced
$inl(a), inr(a)$	$inl(a), inr(a)$	a reduced
$<a,b>$	$<a,b>$	a, b reduced
$\lambda x.b$	$\lambda x.b$	
void	*void*	
int	*int*	
atom	*atom*	
$a < b$	$a < b$	a, b reduced
A *list*	A *list*	A reduced
$A \vert B$	$A \vert B$	A, B reduced
$A \# B$	$A \# B$	A reduced
$x{:}A \# B$	$x{:}A \# B$	A reduced
$A \rightarrow B$	$A \rightarrow B$	A reduced
$x{:}A \rightarrow B$	$x{:}A \rightarrow B$	A reduced
$A//B$	$A//B$	A reduced
$(x,y){:}A//B$	$(x,y){:}A//B$	A reduced
$\{A \vert B\}$	$\{A \vert B\}$	A reduced
$\{x{:}A \vert B\}$	$\{x{:}A \vert B\}$	A reduced
$a = b$ *in* A	$a = b$ *in* A	a, b, A reduced

7.3.1. What is Strictness Analysis?

Evaluating a function application in a lazy fashion requires that the evaluation of the argument be postponed until it is absolutely necessary. This allows needless computation to be avoided, as in the case in which the value returned by the function does not depend on the value of the argument. However, the construction of closures necessitated by the delay in computation can be expensive [Clack and Peyton Jones 85], and the proscription of computation that might be necessary until it is absolutely necessary prevents the simultaneous evalutation of function body and argument. These problems can be alleviated somewhat by an analysis of the term to be computed to determine which of its arguments a function will necessarily evaluate. Such analysis is called *strictness* analysis.

In general determining whether a function is strict in its arguments is undecidable, because of the undecidability of the halting problem. However, certain heuristics based on an abstract analysis of programs have been examined [Mycroft 80, Clack and Peyton Jones 85, Hudak and Young 86]. The particulars of the approaches differ, but all rely on interpreting functions on a two-element domain, with the top element ($\top$) denoting termination and the bottom element ($\perp$) denoting divergence, under the following criterion. Given a function f of one argument and an interpretation function I, if $I(f)(\perp)=\perp$ then the function f evaluates its argument.

7.3.2. *Preduce*--A Parallel Reduction Procedure

Given the previous informal account of strictness analysis, several observations about *reduce* are evident. First, an analyzer term is strict with respect *eval* in its principal arguments, implying that evaluating principal arguments may be done in parallel. Second, a synthesizer term is strict with respect to *reduce* in in its reducible arguments, implying that the reduction of reducible arguments may take place in parallel. These facts suggest the very simple parallel reduction procedure presented in figure 7.3. In the pseudo-code contained in this figure, $e'||e''$ means "compute e' and e'' in parallel."

7.3.3. Towards a More Complete Strictness Analysis

We close this chapter with suggestions for ways in which a more complete strictness may be carried out, and this analysis will in turn suggest more aggressive strategies for evaluating Nuprl terms in parallel.

The research in strictness analysis mentioned above nicely delineates strictness analysis for recursively defined functions over flat domains. However, the presence of higher-order functions and structured values complicates the issue, as does the two-tiered evaluation approach implicit in *reduce* (and in other evaluators [Arvind, Kathail and Pingali 85]). For example, the Nuprl function

$$\lambda x.spread(x;u,v.u)$$

procedure *Peval (t:term)*

  case *t* **is**

  base: **return** *t*

  synthesizer: **return** *t*

  $N(p_1,\ldots,p_l;x_1,\ldots,x_m;t_1,\ldots,t_n)$:

    return *Peval(contract* $N(Peval(p_1)\|\ldots\|Peval(p_l);x_1,\ldots,x_m;t_1,\ldots,t_n)$

end;

procedure *Preduce (t:term)*

  case *t* **is**

  base: **return** *t*

  $C(r_1,\ldots,r_n;x_1,\ldots,x_m,;1,\ldots,t_l)$:

    return $C(Preduce(r_1)\|\ldots\|Preduce(r_n);\ x_1,\ldots,x_m;t_1,\ldots.t_l)$

  $N(p_1,\ldots,p_l;x_1,\ldots,x_m;t_1,\ldots,t_n)$:

      return *Preduce(Peval(t))*

end

Figure 7.3. The *Preduce* Procedure.

calculates the first projection for pairs. Because of the definition of *reduce*, this function may terminate even though *reduce* may not terminate on an argument a; in particular, if a evaluates to a pair $<a_1, a_2>$ and *reduce* terminates on a_1, then this function will terminate on a. In order to handle this situtation the strictness domains must be defined with some care, as the next section demonstrates.

7.3.2.1. Defining the Domains

Since noncanonical forms evaluate rather than reduce their principal arguments, reducing a noncanonical form may terminate even though reducing its principal arguments may not, as we have seen. The reduction of synthesizer terms does not reflect this anomaly, since a term synthesizer is strict with respect to *reduce* in its reducible positions and not strict with respect to either *reduce* or *eval* in its other positions. These facts suggest the definition of two strictness domains, D_R and D_{ER}, that reflect strictness behavior of different terms.

D_R is a two-element domain consisting of a top element, $\top$, to which terms for which *reduce* terminates will be mapped, and a bottom element, $\perp$, to which elements for which *reduce* diverges will be mapped. The term sysnthesizers, then, will be function mapping elements in D_R to D_R.

The structure of D_{ER} is necessarily more complicated, due to the interplay between *eval* and *reduce.* The purpose of this domain is to supply the strictness information of arguments to noncanonical forms, and as such it must contain elements that show the points where *eval* may converge in the computation of a term, even though *reduce* may diverge. Intuitively, the $\perp$ element in D_R (the two-element reduction domain) is carved up into different pieces corresponding to the behavior of the interaction between *eval* and *reduce* on the noncanonical term.

Syntactically, the domain, D_{ER}, of terms may be described by the grammar in figure 7.4; we shall briefly describe the intuition behind the

$$
\begin{array}{lcl}
v & ::= & \perp \\
 & \mid & \top \\
 & \mid & \langle v, v \rangle \\
 & \mid & v.v \\
 & \mid & [v\text{-}v] \\
 & \mid & \lambda x.v
\end{array}
$$

Figure 7.4. The Strictness Domain D_{ER} for Terms.

domain objects here. $\perp$ represents divergence of evaluation; terms for which *eval* diverges will be mapped to $\perp$. $\top$ represents termination of reduction; terms for which *reduce* converges will be mapped to $\top$. The rest of the objects describe the behavior of *eval* in conjunction with *reduce* on associated analyzer terms. The element $\langle v_1, v_2 \rangle$ represents terms that evaluate to pairs, with subterms computing as specified by v_1 and v_2 on the first and second components, respectively, of the pair. The element $v_1.v_2$ represents terms that evaluate to conses, with subterm behavior specified by v_1 and v_2 on the head and tail, respectively, of the cons. The element $[v_1\text{-}v_2]$ represents terms that evaluate to *inl* or *inr* terms, with subterms behaving as specified by v_1 and v_2. Finally $\lambda x.v$ represents terms that evaluate to λ-terms, with v reflecting the strictness properties of the body of the function with respect to the strictness properties of the argument.

Figure 7.5 describes the ordering on terms in D_{ER}. The intent of this

$a \leq b$ if and only if:

(i) $a = \perp$, or

(ii) $b = \top$, or

(iii) $a = <c,d>$, $b = <e,f>$ and $c \leq e$, $d \leq f$, or

(iv) $a = [c\text{-}d]$, $b = [e\text{-}f]$ and $c \leq e$, $d \leq f$, or

(v) $a = c.d$, $b = e.f$ and $c \leq e$, $d \leq f$.

$a = b$ if and only if:

(i) $a = \perp$ and $b = \perp$, or

(ii) $a = \top$ and $b = \top$, $<\top,\top>$, $[\top\text{-}\top]$, or $\top.\top$, or

(iii) $a \leq b$ and $b \leq a$.

Figure 7.5. The Ordering of Terms in D_{ER}.
$a,b,c,d,e,f \in D_{ER}$.

ordering, and of this domain, is that noncanonical terms will map elements of D_{ER} to D_R and that the noncanonical terms will be monotonic with respect to D_{ER}.

7.3.3. Interpreting Nuprl Terms

Given the domains, the task becomes intepreting Nuprl terms. The process is subtle, and beyond the scope of this thesis. However, some remarks are in order.

Clearly, each noncanonical term is strict in its principal arguments with respect to D_{ER}, and each term is also monotonic. It is also the case that a noncanonical term will terminate abnormally if its principal arguments do not compute to expected values. Thus,

$$spread([u\text{-}v];x,y.t) = \top.$$

Furthermore, a structural approach is mandated by the existence of subterms containing binding occurrences of variables; thus, in the above term, the result of

$$spread(<u,v>.x,y.t)$$

will depend on the structure of t.

7.3.4. Calculating Strictness Information

Given the interpretations of Nuprl terms, calculating useful strictness information involves computing the "largest" value in D_{ER} that, when substituted in a principal position in an interpreted term, causes the term to diverge. Thus, for a term like

$$spread(x;u,v.spread(u;u',v'.u'+v'))$$

the desired value is $<<\perp,\perp>,\top>$, reflecting the fact that both subterms of the first component of the outermost pair will be reduced in the final result. For a term like $x+y$, the value to be returned for x is $\perp$ and the value for y is $\perp$. Formally, the value c to be calculated for a term t with free variable x satisfies the following property: $\forall d>c.\ t[d/x]=\top$.

Chapter 8

Conclusions

In broad terms this thesis examines how one might use a system like Nuprl to reason about concurrency. One approach to doing so involves developing a model of concurrency in the Nuprl language and using the Nuprl logic, and logics developed using the Nuprl logic, to reason about the model, thereby employing Nuprl as a metatheory for concurrency. The bulk of the thesis is devoted to examining this kind of approach. Two different models of concurrency are developed independently of the Nuprl system, their expressiveness examined, and their development inside the Nuprl type theory considered. Mention is also made of using the tactic facilities of Nuprl to develop proof strategies for reasoning about concurrency.

This thesis also examines a second approach to using Nuprl to reason about concurrency that involves altering the system itself to support a notion of concurrency. One of the basic tenets of the treatment of Nuprl as a programming language is that proofs and programs are different aspects of the same entity, and as a consequence the Nuprl system supplies a means of extracting sequential programs from proofs of theorems, sequential programs whose correctness is guaranteed by the extraction process. We extend this paradigm, in a way, to concurrent programs by examining a means in which Nuprl terms may be computed in parallel, so that the programs that are extracted from theorems may be viewed as parallel programs in the sense that they execute in parallel.

The remainder of this chapter examines these two approaches in more detail; in particular, we attempt to assess the contribution of this work and to examine potential future directions of research stemming from this work.

8.1. Using Nuprl as a Metatheory

While we cannot yet claim to have developed a tool that is an automatic concurrent program verifier, we have indicated how a theory of formal mathematics, together with an implementation of the theory that incorporates appropriate tools for developing proofs and encapsulating notation and inferences, can be used to reason about concurrency in a rigorous, machine-checked fashion. In doing so we have used semantic-based techniques in which we develop a denotational model of concurrency and then use it to provide accounts of programming language constructs. It is then possible to use the semantic theory as a basis for giving accounts of logics used to reason about concurrency, thereby providing a unified framework for developing and reasoning about programs.

8.1.1. The Semantic Models

The two semantic models given in this thesis, the *Proc* model of chapter 4 and the *ST* model of chapter 5, differ makedly in their structure. *Proc* is trace-based, while *ST* relies on a theory of nonj-well-founded sets for its basis. Milner's CCS is the yardstick we choose to examine the expressiveness of the semantic models; in particular, the notion of process equivalence induced by each model is compared to the CCS equivalences.

The *Proc* model turns out not to be expressive enough to model CCS because the equivalence induced by the model is not as fine as CCS strong equivalence and not comparable with CCS observational equivalence. However, it does have interesting properties in its own right; the CCS program constructors, and the constructors for SCCS and CSP, can be developed in the theory, and a natural semantics can be given to temporal logic using the theory. Although not pursued in detail, it is also relatively straightforward to implement the model in Nuprl; a sketch of such an implementation is presented.

The *ST* model is expressive enough to be a model of CCS, as the equivalence in the model is exactly strong equivalence. The inductive nature of the definition of the model also enables a very straightforward account of the process constructors of CCS and SCCS to be given. This theory relies on a relatively exotic brand of set theory, non-well-founded set theory, for its mathematical justification. The model, however, does provide a natural framework for giving a semantic account of the proof theory of the Hennessy-Milner logic due to Stirling. Chapter 6 details an implementation of the model conducted in Nuprl, in which the CCS terms are defined, the CCS expansion theorem proved, and the correctness of a binary semaphore demonstrated.

8.1.2. Work to be Done

Much work remains to be done on both the theoretical side of using Nuprl as a metatheory and on the practical side of using Nuprl as a system for aiding in reasoning about concurrency. In particular, the kinds of properties that can be stated in the temporal logic described in chapter 4 could be formalized, and the nature of the computation described by the *Proc* model could be more firmly established. In particular, the "one-action lookahead" machines of section 4.3.2 could be formalized, and their relationship with *Proc* determined. The equivalence in *Proc* should also be compared with Milner's observational congruence.

In chapter 6, mention is made of equality in finitary *ST* being decidable. Proving this in Nuprl would yield a decision procedure for equality in finitary *ST*, which in turn would yield a decision procedure for strong equivalence in CCS without recursion. Developing this decision procedure would be very interesting, because equality between CCS processes would would no longer need to be proven; rather, the decision procedure could be invoked. The possibility of extending such a decision procedure to CCS with recursion should also be examined.

On the system side, a more extended example of using Nuprl to reason about CCS needs to be conducted. Also, the power of tactics, although frequently alluded to, is not presented compellingly with an example; a tactic to aid in proofs of deadlock-freedom would be interesting to investigate. An account of a logic also needs to be implemented. Much of the feasibility of all this depends, however, on an improved implementation of Nuprl.

8.2. Introducing Parallelism into the Nuprl Theory

Chapter 7 describes a means of "parallelizing" the evaluation procedure for Nuprl terms. This section re-examines this procedure, suggests ways it may be extended, and then considers other ways in which concurrency may be incorporated into the Nuprl theory.

8.2.1. Parallel Evaluation

In order to introduce a parallel evaluation procedure we first developed a more eager evaluation procedure that respected the semantics of the Nuprl type theory. This procedure was then "parallelized" in a very straightforward fashion by essentially requiring that recursive calls to the procedure that occur at the same level be computed concurrently. We also discuss informally a framework for implementing a more aggressive scheme for parallelization based on strictness analysis of Nuprl terms.

The work on strictness analysis is incomplete, due to the difficulty of the problem and to the complexity of the Nuprl term language, and it needs to be finished. Completing the strictness analysis would be done best in two stages. The first stage would involve paring down the term language to be studied; an appropriate candidate might be a language of lists, where the only two term constructors allowed would be a cons operator and lambda abstraction. After this analysis is done, the second stage would involve extending the work to the full Nuprl language.

Although we do not address it, there is another means of introducing parallel evaluation into Nuprl, and this approach could also be developed in detail. In this evaluation scheme, redexes are computed in a breadth-first fashion within a term until a reduced form is reached, at which point the procedure terminates. In this scheme, it is possible that unnecessary computation will be performed, since, for example, all four subterms of an *int_eq* term may be computed, even though only three of the four are necessary. However, this scheme is conceptually straightforward and adaptable to a variety of parallel architectures.

The practical details of implementing a parallel evaluation scheme are not addressed in this thesis. Much work has been done on the practical aspects of evaluating functional programming languages, and adapting this work to the Nuprl environment would yield useful results.

The next two sections examine other ways in which concurrency could be incorporated into Nuprl.

8.2.2. Concurrent Types

Although parallel evaluation allows the implicit introduction of concurrency into the Nuprl theory it does not allow parallelism to be explicitly introduced into terms by users of the theory. One means of addressing this shortcoming involves introducing a special type constructor, a *concurrent* type constructor, into the theory. Intuitively, inhabitants of concurrent types would be communicating processes that execute in parallel. Proof rules associated with this type constructor would specify how objects of these types may be built, and the programs extracted from proofs would be concurrent, communicating programs.

One way to imagine the nature of these types is to consider them as generalizations of the dependent product type. In the type $x{:}A\#B$ one may think of type A as "sending" information to expression B via variable x. In principle, there is no reason why information should not flow both ways, with variables being bound to subtypes in one expression and appearing free in the other expression. Generalizing the proof rules for dependent product might prove to be the best way to develop proof rules of types having this form.

8.2.3. Concurrent Tactics

Another area in which concurrency might be of practical benefit to the Nuprl system is the metalanguage of the system. Currently, tactics are written in a sequential programming language, ML. However, the structure of proof trees in Nuprl suggests several ways in which concurrency may be introduced into tactics. For example, the *THEN* tactical effects sequential composition of tactics; the tactic t_1 *THEN* t_2 applies tactic t_1 to the current goal and then applies tactic t_2 to each of the children resulting from the execution of t_1. Clearly, t_2 could be applied concurrently to each of these children. Similarly, the *ORELSE* tactical provides a means of catching tactic failure. The tactic t_1 *ORELSE* t_2 applies t_1 to a goal, and if t_1 fails it then applies t_2. One can imagine having t_1 and t_2 execute concurrently on the goal, with the result of t_1 being instantiated if it succeeds and the result of t_2 being instantiated otherwise. It would even be possible to alter the semantics of *ORELSE* in such a way that the result that is instantiated would be the result of the first tactic to terminate normally. The work of Knoblock [Knoblock and Constable 86, Knoblock 87] on reflecting the Nuprl metalanguage into Nuprl, in conjunction with a parallel evaluation procedure like the one developed in this thesis, would be one way in which concurrency could be introduced into tactics.

On a larger scale, principles of communicating processes may be applied to the tactic setting to create large networks of co-operating tactics. Such a network philosophy would be useful in developing large theorem-provers in the context of Nuprl; one could imagine having a network consisting of tactics doing arithmetic and tactics doing logical deductions and tactics doing term-rewriting in a system to prove theorems about number theory.

8.3. Implementing Set Theory

In the course of our implementation of models of concurrency we develop a substantial portion of finite set theory. This section reviews our implementation and points out areas in which future work can be done.

8.3.1. The Implementation of *finset*

The implementation described in chapter 6 of a finite set type constructor for U_1 types is array-based. The finite sets of type T are represented as arrays over type T with equivalence between arrays redefined, via the quotient type constructor, as extensional set equality. Several of the sets and operations of set theory are described, including the null set, membership, union, decidable comprehension, collection and singletons, and the axioms of set theory are shown to hold of the implementations we provide. Tactics are provided that implement some of the axioms of set theory as inference rules, thereby allowing Nuprl to be used as an environment for reasoning about finite sets.

One immediate area in which the implementation can be extended involves the powerset construction. We do not define a powerset operator, primarily because we do not need it in order to define CCS. In principle, however, a powerset operator can be developed and proven to satisfy the appropriate axiom of set theory.

The theory we have developed can also be used to develop a set theory having no base type. Consider the type defined as follows.

$$FiniteSets = rec(\ fs.\ finset(fs))$$

FiniteSets is a U_1 type whose elements are finite sets of finite sets. $\varnothing$ (with implicit base type *void*) is in *FiniteSets*, as is $\{\varnothing\}$, $\{\varnothing, \{\varnothing\}\}$, and so forth. Pursuing this development of set theory could be interesting because equality in this theory is decidable, meaning that set membership is also decidable; this implies the existence of interesting decision procedures that could be developed and proven correct in Nuprl. Another interesting exercise would be to see if the regularity axiom of ZF set theory is provable in Nuprl using this implementation of sets.

Having developed finite set theory, the next natural theory to examine in Nuprl is general constructive set theory. One way to do so would be to identify sets with images of functions over the positive integers, thereby allowing the existence of infinite sets. Several of the operations defined for finite sets are straightforward to define; a union operator could take two sets and produce their interleaving as its result, and collection would just be function composition. Membership would also be easy to define. However, there is no clear way to define the null set, and representing a comprehension scheme would also be subtle. The Ph.D. thesis of McCarty [McCarty 84] could provide some insights into how to define these.

In the context of the *inf* types of Mendler et al. [Mendler, Panangaden and Constable 86] it becomes possible to develop "finite" non-well-founded finite set theory using the *finset* constructor. Consider the type defined as follows.

$$NWSets = inf(st.finset(st))$$

NWSets consists of the sets with finite numbers of elements; however, these sets may also be non-well-founded, so that it would be possible to define a set X with $X \in X$.

Appendix

The Nuprl Library

This appendix contains a listing of the Nuprl library described in chapter 6. The listing has been edited somewhat in order to improve its readability, but what appears is essentially an exact replica of the library. The library took approximately four months of full-time effort to build and contains approximately 240 objects, about 230 of which appear here (the other objects were subsequently not used in the development and so have been omitted).

In general, the listing of a Nuprl object contains four pieces, the first three of which appear on the first line of the listing. The first piece is an asterisk (*) indicating that the object has been checked by the Nuprl system and found to be complete. If the object is a theorem, this means that the theorem has been proved; if it is a definition then the definition has been parsed; if it is an ML object then the ML definitions have been parsed, type-checked and added to the ML environment; if it is an evaluation object then the evaluations have been carried out and the bindings added to the evaluation environment. The second part of the listing describes the kind of the object. DEF signifies a definition, THM signifies a theorem, ML signifies an ML object, and EVAL signifies an evaluation object. The third part is the object's name.

The final part of the listing of an object appears indented below the other three parts described above; its contents depends on the kind of the object. In the case of a definition, the display form of the definition (that is, what appears on the screen when the definition is invoked) is listed, followed by "==" and by the notation that the display form corresponds to. If the object is a theorem then the statement of the theorem appears after a ">>" and is followed by the term that is extracted from the proof of the theorem (see chapter 3 for more on extraction). If the object is an ML object then the ML code contained in the object is printed; this code includes tactics, which generally have the first letters of their names capitalized, and auxiliary functions and bindings used by tactics. If the object is an evaluation object then a listing of the bindings defined in the object appears.

A final note about the organization of the library is in order. For technical convenience many of the set operations (*union*, for example) contain alternative formulations (*alt_union*, in the case of *union*). The names of these alternative formulations are always preceded by *alt*. These alternative formulations are the same as other formulations except that the type parameter that appears at the beginning of each set operation is omitted.

```
* DEF DEFS

     ************************************
     General purpose DEFs
     ************************************==


* DEF c

     (*<c:comment>*)==


* DEF t

     -- <T:tactic>==(<T>) THEN Autotactic


* DEF tw

     (<T:tactic> ... {wf})==(<T>)
         THEN (IfThen is_wf_goal Autotactic)


* DEF thenwf

     <T:tactic> THENWF <U:tactic>==
     (<T>) THEN (IfThen is_wf_goal (<U>))


* DEF thenseq

     Sequence <t:term> THEN-TO-SEQUAND <T:tactic>==
     Seq ['<t>'] THENL [<T>;Idtac]


* DEF na
```

```
<r:rule> without Autotactic==
ref ‘<r>‘
```

* DEF tc

```
--* <T:tactic>==(<T>) THEN Try (Complete Autotactic)
```

* DEF tm

```
--ϵ <T:tactic>==(<T>) THENW Autotactic
```

* DEF notree

```
<T:tactic>° without display maintenance==
(apply_without_display_maintenance <T>)
```

* DEF nothing

```
<a:anything>==
```

* DEF squash

```
↓ <T:type>=={(0 in int)|(<T>)}
```

* DEF spread

```
let <x:var>,<y:var>=<t:term> in <tt:term>
==spread(<t>;<x>,<y>.<tt>)
```

* DEF dblspread

```
let <w:var>,<x:var>,<y:var>,<z:var>=
<t1:term>,<t2:term> in <t3:term>==
let <w>,<x>=<t1> in let <y>,<z>=<t2> in <t3>
```

* DEF and

```
<P:prop>&<Q:prop>==((<P>)#(<Q>))
```

* DEF or

```
<P:prop>∨<Q:prop>==((<P>)|(<Q>))
```

* DEF imp

```
<P:prop>=><Q:prop>==((<P>)->(<Q>))
```

* DEF not

```
¬<P:prop>==<P>=>void
```

* DEF iff

```
<P:prop>\<=><Q:prop>==<P>=><Q>&<Q>=><P>
```

* DEF all

```
∀<x:var>:<T:type>.<P:prop>==(<x>:(<T>)->(<P>))
```

```
* DEF all2

      ∀<x:var>,<y:var>:<T:type>.<P:prop>
      ==∀<x>:<T>.∀<y>:<T>.<P>


* DEF all3

      ∀<x:var>,<y:var>,<z:var>:<T:type>.<P:prop>
      ==∀<x>:<T>.∀<y>,<z>:<T>.<P>


* DEF all4

      ∀<w:var>,<x:var>,<y:var>,<z:var>:<T:type>.<P:prop>
      ==∀<w>:<T>.∀<x>,<y>,<z>:<T>.<P>


* DEF some

      ∃<x:var>:<T:type>.<P:prop>==(<x>:(<T>)#(<P>))


* DEF some2

      ∃<x:var>,<y:var>:<T:type>.<P:prop>
      ==∃<x>:<T>.∃<y>:<T>.<P>


* DEF some3

      ∃<x:var>,<y:var>,<z:var>:<T:type>.<P:prop>
      ==∃<x>:<T>.∃<y>,<z>:<T>.<P>


* DEF true
```

```
true==(0 in int)
```

* DEF set

```
{<x:var>:<T:type>|<P:prop>}==({<x>:(<T>)|(<P>)})
```

* DEF discrete

```
discrete(<T:type>)==
∀x,y:T.x=y in T∨¬x=y in T
```

* DEF p1

```
<x:tuple>.1==spread(<x>;u,v.u)
```

* DEF p2

```
<x:tuple>.2==spread(<x>;u,v.v)
```

* DEF if_then

```
if <p:prop> then <e1:exp> else <e2:exp>==
decide(p;u.e1;u.e2)
```

* DEF l

```
λ<x:var>.<t:term>==(\ <x>.(<t>))
```

* DEF l2

```
    λ<x:var><y:var>.<t:term>==λ<x>.λ<y>.<t>

* DEF cons

    <h:A>^<t:A list>==((<h>).(<t>))

* DEF ap

    <f:term><x:term>==((<f>)(<x>))

* DEF le

    <x:Int>≤ <y:Int>==¬ (<y>)<(<x>)

* DEF lele

    <x:Int>≤ <y:Int>≤ <z:Int>==<x>≤ <y>&<y>≤ <z>

* DEF lle

    <x:Int>\<<y:Int>≤ <z:Int>==(<x>)<(<y>)&<y>≤ <z>

* DEF lel

    <x:Int>≤ <y:Int>\<<z:Int>==<x>≤ <y>&(<y>)<(<z>)

* DEF ll

    <x:Int>\<<y:Int>\<<z:Int>==(<x>)<(<y>)&(<y>)<(<z>)
```

* THM Int_abs_

```
>> Int -> Int

Extraction:
λ n. less(n;0;-n;n)
```

* DEF Int_abs

```
|<n:Int>|==term_of(Int_abs_)<n>
```

* THM nat_

```
>>int->int

Extraction:
λi.less(i;0;0;i)
```

* DEF nat

```
↑<i:Int>==
term_of(nat_)(<i>)
```

* DEF eq

```
<m:Int>=<n:Int>==((<m>)=(<n>) in Int)
```

* DEF neq

```
<a:term>≠ <b:term> in <T:type>==
```

```
¬<a>=<b> in <T>
```

* THM interval_

```
>>int->U1

Extraction:
λx.{i:int|0<i≤ x}
```

* DEF interval

```
{1,...,<n:int>}==term_of(interval_)(<n>)
```

* THM decidable_

```
>>T:U1->P:(T->U1)->U1

Extraction:
λT.λP.∀a:T.P(a)∨¬P(a)
```

* DEF decidable

```
<P:predicate> is decidable on <T:type>==
term_of (decidable_)(<T>)(<P>)
```

* DEF mono

```
*************************************
General Monotonicity, Interval and Range Theorems
*************************************==
```

* THM monotonicity1

```
>>∀i,j,k:int.i<j=>i+k<j+k
```

* THM monotonicity2

```
>>∀i,j:int.i<j=>0<j-i
```

* THM monotonicity3

```
>>∀i,j,k:int.i=j=>i+k=j+k
```

* THM monotonicity4

```
>>∀i,j,k:int.i≤ j=>i+k≤ j+k
```

* THM monotonicity5

```
>>∀i,j,k,l:int.i<j=>k<l=>i+k<j+l
```

* THM monotonicity6

```
>>∀i,j,k,l:int.i≤ j=>k≤ l=>i+k≤ j+l
```

* THM monotonicity7

```
>>∀i,j,k,l:int.i<j=>k=l=>i+k<j+l
```

* THM monotonicity8

```
>>∀i,j,k,l:int.i≤ j=>k=l=>i+k≤ j+l
```

* THM lle_to_l

```
>>∀i,j,k:int.i<j≤ k=>i<k
```

* THM lle_left

```
>>∀i,j,k,l:int.i<j≤ k=>l≤ i=>l<j≤ k
```

* THM lle_right

```
>>∀i,j,k,l:int.i<j≤ k=>k≤ l=>i<j≤ l
```

* THM lle_void

```
>>∀i,j,k:int.i<j≤ k=>k≤ i=>void
```

* THM lle_plus_positive

```
>>∀i,j,k,l:int.i<j≤ k=>0≤ l=>i<j+l≤ k+l
```

* THM lle_minus

```
>>∀i,j,k,l:int.i<j≤ k=>i-l<j-l≤ k-l
```

* THM lle_minus_simplify

```
>>∀i,j,k:int.i<j≤ i+k=>0<j-i≤ k
```

*** THM lle_to_llenat**

>>∀i,j,k:int.i<j≤ k=>i<j≤ ↑ k

*** THM nat1**

>>∀i:int.i≤ 0=>↑ i=0 in int

*** THM nat2**

>>∀i:int.0<i=>↑ i=i in int

*** THM nat1_1**

>>∀i:int.↑ i=0 in int=>i≤ 0

*** THM nat2_2**

>>∀i:int.0<↑ i=>↑ i=i in int

*** THM nat3**

>>∀i:int.0≤ ↑ i

*** THM nat4**

>>∀i:int.i≤ ↑ i

*** THM natset1**

```
>>∀i,j:int.∀p:{1,...,↑ i+↑ j}.p<↑ i+1=>0<p≤ i
```

*** THM natset2**

```
>>∀i,j:int.∀p:{1,...,↑ i+↑ j}.↑ i+1≤ p=>0<p-↑ i≤ j
```

*** THM split_natrange**

```
>>∀i,j,k:int.0<i≤ ↑ j+↑ k=>0<i≤ ↑ j∨↑ j<i≤ ↑ j+↑ k
```

*** THM range_to_natrange**

```
>>∀i,j:int.0<i≤ j=>0<i≤ ↑ j
```

*** THM natrange_to_range**

```
>>∀i,j:int.0<i≤ ↑ j=>0<i≤ j
```

*** THM interval_to_sumnatinterval**

```
>>∀i,j:int.∀k:{1,...,i}.k in {1,...,↑ i+↑ j}
```

*** THM Interval_to_sumnatinterval**

```
>>∀i,j:int.∀k:{1,...,j}.k+↑ i in {1,...,↑ i+↑ j}
```

*** THM simplify_set_equality**

```
>>∀i,j:int.∀k:{1,...,i}.(k+j)-j=k in {1,...,i}
```

* DEF FiniteSets

```
*************************************
Finite Sets
*************************************==
```

* THM set_eq_

```
>>T:U1->(n:int#{1,...,n}->T)->(n:int#{1,...,n}->T)->U1

Extraction:
λT.λs.λt.
∀i:{1,...,s.1}.↓ ∃j:{1,...,t.1}.s.2(i)=t.2(j) in T&
 ∀i:{1,...,t.1}.↓ ∃j:{1,...,s.1}.t.2(i)=s.2(j) in T
```

* DEF set_eq

```
<s:finset> =e <t:finset> over <T:U1> ==
term_of(set_eq_)(<T>)(<s>)(<t>)
```

* THM set_eq_reflexive

```
>>∀T:U1.∀s:(n:int#{1,...,n}->T).s =e s over T
```

* THM set_eq_symmetric

```
>>∀T:U1.∀s,t:(n:int#{1,...,n}->T).
s =e t over T =>t =e s over T
```

* THM set_eq_transitive

```
>>∀T:U1.∀r,s,t:(n:int#{1,...,n}->T).
r =e s over T =>s =e t over T =>r =e t over T
```

*** THM finset_**

```
>>U1->U1

Extraction:
λT.(s,t):(n:int#{1,...,n}->T)//s =e t over T
```

*** DEF finset**

```
finset(<T:type>)==
term_of(finset_)(<T>)
```

*** ML finset_tactics**

```
letrec hyp_count_id l v i =
   if l=[] then fail
   if v = fst(destruct_declaration (hd l)) then i
   else hyp_count_id (tl l) v i+1;;

let hyp_no_for_id hyplist v =
   hyp_count_id hyplist v 1;;

letrec second_to_last_id l =
   if tl (tl (tl l)) = [] then
      fst (destruct_declaration (hd l))
   else second_to_last_id (tl l);;

letrec second_to_last_rec l i =
   if tl (tl (tl l)) = [] then i
   else second_to_last_rec (tl l) i+1;;
```

```
let second_to_last l =
   second_to_last_rec l 1;;

letrec next_to_last_rec l i =
   if tl (tl l) = [] then i
   else next_to_last_rec (tl l) i+1;;

let next_to_last l =
   next_to_last_rec l 1;;

letrec last_rec l i =
   if tl l = [] then i
   else last_rec (tl l) i+1;;

let last l =
   last_rec l 1;;

let ElimFinset v = \p.(
   (ComputeHypType (hyp_no_for_id (hypotheses p) v)
   THEN Elim (hyp_no_for_id (hypotheses p) v)
   THENL
    [Autotactic
    ;COMPLETE(Autotactic)
      ORELSE COMPLETE(Autotactic
       THEN -- Seq ['finset(T) in U17']
       THEN \p.(NormalizeHyp (last (hypotheses p))
                THEN NormalizeConcl THEN Autotactic) p)
      ORELSE Idtac
    ;\q.(-- BringDependingHyps (second_to_last_id
                                  (hypotheses q))
         THEN Elim (second_to_last (hypotheses q))
         THEN Elim (next_to_last (hypotheses q))
         THEN Intro
         THENL [Idtac; Autotactic]
         ) q
    ]) p)
```

```
;;

let FinsetTypeWellFormed q =
   let equands = fst (destruct_equal (concl q)) in
   let quot = hd(equands) in
   let basetype =
       fst(snd(snd(destruct_quotient quot))) in
   let elementfunctiontype =
       snd(snd(destruct_product basetype)) in
   let elementtype =
       snd(snd(destruct_function elementfunctiontype)) in

   let finsettype =
       make_apply_term
          (make_term_of_theorem_term(`finset_`))
          (elementtype) in
   let wfsequand =
       make_equal_term('U1')([finsettype]) in
   (Try(-- ref `cumulativity`) THEN
   -- Seq [wfsequand]
   THEN \p.(NormalizeHyp (last (hypotheses p))
            THEN NormalizeConcl THEN Autotactic) p) q
;;
```

```
* ML finset_tactics_2

      let SplitInterval i p =
         ((Elim i ... {wf})
         THEN (\q.LemmaFromHyps `split_natrange`
                  [(last (hypotheses q))] [] q)
         THEN Try (Complete Autotactic)
         THEN (\q.(Elim (last (hypotheses q)) q))
         THEN (\q.Thinning [i;(second_to_last (hypotheses q));
                               (next_to_last (hypotheses q))] q)

         ) p
```

```
;;

let is_finset_equality_goal p=
   let c = conclusion p in
      if (is_equal_term c)
      then if (is_apply_term (snd (destruct_equal c)))
           then (fst (destruct_apply
                         (snd (destruct_equal c))))
                =’term_of(finset_)’
           else 0=1
      else (is_equal_term c)
;;

let IntervalTypeWellFormed =
   EqualityIntro
;;

letrec declno i l =
   if i=1 then hd(l)
   else declno (i-1) (tl l)
;;

let SetEqualityIntroUsingHyp i p =
   (TopLevelComputeHyp i THEN TopLevelComputeConcl
    THEN ReduceHyp i THEN ReduceConcl
    THEN RestoreDefRefsInHyp i
    THEN RestoreDefRefsInConcl
    THEN Elim i THEN Thinning [i]
    THEN (RepeatFor 2 Intro)
    THENL
      [ \q.let l = hypotheses q in
           let n = last l in
           (ElimOn (n-2) (make_var_term
                          (id_of_declaration (declno n l)))

           THENL
           [ ref ‘equality‘
```

```
          ; Elim (n+1) THEN Thinning [n+1;n+2]
            THENL
            [ Idtac
            ; ref `explicit intro axiom`
              THEN MemberIntro
              THENL
              [ Idtac
              ; Elim (n+1) THEN Thinning [i;i+1;n+1]
              ]
            ]
          ]) q
     ; IntervalTypeWellFormed
     ; \q.let l = hypotheses q in
          let n = last l in
          (ElimOn (n-1) (make_var_term
                         (id_of_declaration (declno n l)))

          THENL
          [ ref `equality`
          ; Elim (n+1) THEN Thinning [n+1;n+2]
            THENL
            [ Idtac
            ; ref `explicit intro axiom`
              THEN MemberIntro
              THENL
              [ Idtac
              ; Elim (n+1) THEN Thinning [i;i+1;n+1]
              ]
            ]
          ]) q
     ; IntervalTypeWellFormed
     ]
  ) p
;;
```

```
* DEF tf

    --f <t:tactic>==
    (<t> THEN Autotactic) THEN
    (IfThen is_finset_equality_goal
            (Repeat (EqualityIntro THEN Autotactic))


* DEF tcf

    --*f <t:tactic>==
    (<t> THEN Try(Complete(Autotactic))) THEN
    (IfThen is_finset_equality_goal
            (Repeat (EqualityIntro THEN Autotactic))


* DEF Membership

    ************************************
    Definition and axiomatization of Set Membership
    *************************************==


* THM in_

    >>T:U1->a:T->s:finset(T)->U1

    Extraction:
    λT.λa.λs.↓ ↓ ∃i:{1,...,s.1}.s.2(i)=a in T


* DEF in

    <a:element> ϵ<T:type> <s:finset> ==
    term_of(in_)(<T>)(<a>)(<s>)
```

* THM extensionality

```
>>∀T:U1.∀s,t:finset(T).↓ (∀a:T.a ϵT s <=>a ϵT t )<=>s=t in
finset(T)
```

* ML finset_tactics_3

```
let FinsetExtensionality p =
   let equands,type = destruct_equal (conclusion p) in
   let BaseTypeOfFinset = snd (destruct_apply type) in
   (InstantiateLemma `extensionality`
      [BaseTypeOfFinset ; hd equands ; hd (tl equands)]
   THEN
   \q.(if conclusion p = conclusion q
       then
       Elim (last (hypotheses q))
       THENL
       [Idtac
       ;\r.(Elim (last (hypotheses r))
            THEN
            \s.(Elim (next_to_last (hypotheses s))
               THENL
               [\t.let lhyp=last (hypotheses t) in
                 (Thinning [lhyp-4;lhyp-3;
                            lhyp-2;lhyp-1;lhyp]
                 THEN RepeatFor 2 Intro) t
               ;\tt.let lhyp=last (hypotheses tt) in
                (Thinning [lhyp-4;lhyp-3;
                           lhyp-2;lhyp-1]) tt
               ]) s
            ) r
       ]
       else Idtac) q
  ) p
;;
```

* ML finset_tactics_4

```
let ElimSetMembership i =
   Elim i THENL
      [Try (AbsNormalizeConcl
            THEN RestoreDefRefsInConcl)
      ;
       (\p.Elim (last (hypotheses p)) p) THENL
         [Try (AbsNormalizeConcl
               THEN RestoreDefRefsInConcl)
         ;
          \q. let lhyp=last (hypotheses q) in
               Thinning [i;lhyp-3;lhyp-2;lhyp-1] q
         ]
      ]
;;
```

* THM in_intro_axiom_lemma

```
>>∀T:U1.∀s:finset(T).∀a:T.a ϵT s =>axiom in a ϵT s
```

* DEF NullSet

```
*************************************
Definition and axiomatization of Null Set
**************************************==
```

* THM null_

```
>>T:U1->finset(T)

Extraction:
λT.<0,λx.x>
```

```
* DEF null

      ∅<T:type>==
      term_of(null_)(<T>)


* THM null_axiom

      >>∀T:U1.∀a:T.¬a ∈T ∅T


* DEF alt_null

      ∅==
      <0,λx.x>


* THM alt_null_

      >>∀T:U1.∅ in finset(T)

      Extraction:
      \T.axiom


* DEF SingletonSet

      ************************************
      Definition and axiomatization of singleton sets
      ************************************==


* THM singleton_

      >>T:U1->a:T->finset(T)
```

```
Extraction:
λT.λa.<1,λx.a>
```

```
* DEF singleton

      {<a:element>}<T:type>==
      term_of(singleton_)(<T>)(<a>)
```

```
* THM singleton_axiom_1

      >>∀T:U1.∀a,b:T.a=b in T=>a ϵT {b}T
```

```
* THM singleton_axiom_2

      >>∀T:U1.∀a,b:T.a ϵT {b}T =>a=b in T
```

```
* DEF alt_singleton

      {<a:element>}==
      λa.<1,λx.a>(<a>)
```

```
* THM alt_singleton_

      >>∀T:U1.∀a:T.{a} in finset(T)

      Extraction:
      \T.\a.(\v0.(\v1.axiom)(v0(a)))
      (term_of(alt_singleton_)(T))
```

```
* DEF Union
```

```
*************************************
Definition and axiomatization of binary union
**************************************==
```

*** THM union_**

```
>>T:U1->finset(T)->finset(T)->finset(T)

Extraction:
λT.λs.λt.
<↑ s.1+↑ t.1,λi.less(i;↑ s.1+1;s.2(i);t.2(i-↑ s.1))>
```

*** DEF union**

```
<S:finset> ∪<T:type> <U:finset>==
term_of(union_)(<T>)(<S>)(<U>)
```

*** THM union_axiom_1**

```
>>∀T:U1.∀s,t:finset(T).∀a:T.a ϵT s ∪T t =>
↓ a ϵT s ∨a ϵT t
```

*** THM union_axiom_2**

```
>>∀T:U1.∀s,t:finset(T).∀a:T.
a ϵT s ∨a ϵT t =>a ϵT s ∪T t
```

*** DEF alt_union**

```
<s:finset>∪<t:finset>==
λst.
```

```
<↑ s.1+↑ t.1,
  λi.less(i;↑ s.1+1;s.2(i);t.2(i-↑ s.1))>(<s>)(<t>)
```

* **THM** alt_union_

```
>>∀T:U1.∀s,t:finset(T).s∪t in finset(T)

Extraction:
\T.\s.\t.(\v0.(\v1.(\v2.axiom)(v1(t)))(v0(s)))
(term_of(alt_union_)(T))
```

* **DEF** BaseLemmas

```
*************************************
Lemmas involving operations on the basetype
*************************************==
```

* **THM** null_lemma

```
>>∀T:U1.∀b:n:int#{1,...,n}->T.b.1≤ 0=>b=∅T in finset(T)
```

* **THM** function_restriction

```
>>∀T:U1.∀i,j:int.j<i=>∀f:{1,...,i}->T.f in {1,...,j}->T
```

* **THM** contains_lemma

```
>>∀T:U1.∀i:int.∀f:{1,...,i}->T.∀j:int.∀g:{1,...,j}->T.
∀k:{1,...,i}.↓ ∃l:{1,...,j}.f(k)=g(l) in T=>
∀k:{1,...,i-1}.↓ ∃l:{1,...,j}.f(k)=g(l) in T
```

```
* THM in_basetype_lemma

      >>∀T:U1.∀s:n:int#{1,...,n}->T.∀a:T.∃i:{1,...,s.1}.
      s.2(i)=a in T in U1


* THM union_basetype_lemma

      >>∀T:U1.∀p,q:n:int#{1,...,n}->T.
      p ∪T q in n:int#{1,...,n}->T


* THM singleton_basetype_lemma

      >>∀T:U1.∀a:T.{a}T in n:int#{1,...,n}->T


* THM null_basetype_lemma

      >>∀T:U1.∅T in n:int#{1,...,n}->T


* DEF Comprehension

      *************************************
      Definition and axiomatization of a comprehension scheme
      *************************************==


* THM comprehension_basetype_lemma

      >>∀T:U1.∀i:int.∀f:{1,...,i}->T.
      ∀P:T->U1.∀p:P is decidable on T.
      ind(i;u,v.∅T;∅T;u,v.decide(p(f(u));w.{f(u)}T ∪T v;x.v))
      in n:int#{1,...,n}->T
```

* THM comprehension_wf_lemma

```
>>∀T:U1.∀i:int.∀f:{1,...,i}->T.
∀P:T->U1.∀p:P is decidable on T.
ind(i;u,v.∅T;∅T;u,v.decide(p(f(u));w.{f(u)}T ∪T v;x.v))
in finset(T)
```

* THM comprehension_inclusion_lemma

```
>>∀T:U1.∀P:T->U1.∀p:P is decidable on T.
∀i:int.∀f:{1,...,i}->T.
∀j:int.∀g:{1,...,j}->T.∀k:{1,...,i}.↓∃l:{1,...,j}.
f(k)=g(l) in T=> ∀a:T.
a ϵT (ind(i;u,v.∅T;∅T
           ;u,v.decide(p(f(u));w.{f(u)}T ∪T v;x.v)))
=> a ϵT (ind(j;u,v.∅T;∅T
           ;u,v.decide(p(g(u));w.{g(u)}T ∪T v;x.v)))
```

* THM comprehension_

```
>>T:U1->P:(T->U1)->
P is decidable on T->finset(T)->finset(T)

Extraction:
λT.λP.λp.λs.
ind(s.1;u,v.∅T;∅T
       ;u,v.decide(p(s.2(u));w.{s.2(u)}T ∪T v;x.v))
```

* DEF comprehension

```
{xϵ<s:finset>|<P:predicate> & <p:decidability>}<T:type>==

term_of(comprehension_)(<T>)(<P>)(<p>)(<s>)
```

* THM comprehension_basetype_lemma_2

```
>>∀T:U1.∀i:int.∀f:{1,...,i}->T.
∀P:T->U1.∀p:P is decidable on T.
{xϵ<i,f>|P & p}T in n:int#{1,...,n}->T
```

* THM comprehension_axiom_1

```
>>∀T:U1.∀P:T->U1.∀p:P is decidable on T.∀s:finset(T).
∀a:T.a ϵT s & P(a) => a ϵT {xϵs|P & p}T
```

* THM comprehension_axiom_2

```
>>∀T:U1.∀P:T->U1.∀p:P is decidable on T.∀s:finset(T).
∀a:T.a ϵT {xϵs|P & p}T => a ϵT s & P(a)
```

* DEF alt_comprehension

```
{xϵ<s:finset>|<P:predicate> & <p:decidability>}==
ind(<s>.1;u,v.∅;∅;u,v.decide(<p>(<s>.2(u));w.{<s>.2(u)}∪v;x.v))
```

* THM alt_comprehension_

```
>>∀T:U1.∀P:T->U1.∀p:P is decidable on T.∀s:finset(T).
{xϵs|P & p} in finset(T)
```

Extraction:

```
\T.\P.\p.\s.
(\v0.(\v1.(\v2.(\v3.axiom)(v2(s)))(v1(p)))(v0(P)))
(term_of(alt_comprehension_)(T))
```

* DEF Set_union

```
*************************************
Definition and axiomatization of arbitrary union
*************************************==
```

* THM set_union_wf_lemma

```
>>∀T:U1.∀i:int.∀f:{1,...,i}->finset(T).
ind(i;u,v.∅T;∅T;u,v.f(u) ∪T v) in finset(T)
```

* THM set_union_inclusion_lemma

```
>>∀T:U1.∀i:int.∀f:{1,...,i}->finset(T).
∀j:int.∀g:{1,...,j}->finset(T).
∀k:{1,...,i}.↓ ∃l:{1,...,j}.f(k)=g(l) in finset(T)
=>∀a:T.a ∈T ind(i;u,v.∅T;∅T;u,v.f(u) ∪T v)
=>a ∈T ind(j;u,v.∅T;∅T;u,v.g(u) ∪T v)
```

* THM set_union_

```
>>T:U1->finset(finset(T))->finset(T)

Extraction:
λT.λs.ind(s.1;u,v.∅T;∅T;u,v.s.2(u) ∪T v)
```

* DEF set_union

```
∪<T:type>(<s:finset-of-finset>)==
term_of(set_union_)(<T>)(<s>)
```

* THM set_union_axiom_1

```
>>∀T:U1.∀s:finset(finset(T)).∀a:T.
(∃t:finset(T).t ϵfinset(T) s &a ϵT t )=>a ϵT ∪T(s)
```

* THM set_union_axiom_2

```
>>∀T:U1.∀s:finset(finset(T)).∀a:T.
a ϵT ∪T(s) =>↓ ∃t:finset(T).t ϵfinset(T) s &a ϵT t
```

* DEF Collection

```
****************************************
Definition and axiomatization of a collection scheme
**************************************==
```

* THM collection_

```
>>T:U1->U:U1->f:(T->U)->finset(T)->finset(U)

Extraction:
λT.λU.λf.λs.<s.1,λx.f(s.2(x))>
```

* DEF collection

```
map <f:function>(<T:type>:<U:type>) on <s:finset>==
term_of(collection_)(<T>)(<U>)(<f>)(<s>)
```

* THM collection_axiom_1

```
>>∀T,U:U1.∀f:T->U.∀s:finset(T).∀a:T.
```

```
a ϵT s =>f(a) ϵU map f(T:U) on s
```

* THM collection_axiom_2

```
>>∀T,U:U1.∀f:T->U.∀s:finset(T).∀b:U.
b ϵU map f(T:U) on s =>↓ ∃a:T.f(a)=b in U & a ϵT s
```

* DEF alt_collection

```
map <f:function> on <s:set>==
<<s>.1,λx.<f>(<s>.2(x))>
```

* THM alt_collection_

```
>>∀T,U:U1.∀f:T->U.∀s:finset(T).(map f on s) in finset(U)
```

```
Extraction:
\T.\U.\f.\s.
(\v0.(\v1.(\v2.(\v3.axiom)(v2(s)))(v1(f)))(v0(U)))
(term_of(alt_collection_)(T))
```

* EVAL eval_set

```
let on =
λa.λl.
list_ind(l;inr(λx.x);h,t,i.int_eq(a;h;inl(axiom);i));;

let ifon =
λa.λl.λt1.λt2.decide(on(a)(l);x.t1;y.t2);;

let build_set =
λl.list_ind(l;∅int;h,t,i.{h}int ∪int i);;
```

```
let list_set =
λs.ind(s.1;u,v.nil;nil;u,v.ifon(s.2(u))(v)(v)(s.2(u).v));;


let null = ∅int;;

let singleton = λi.{i}int;;

let join = λst.s ∪int t;;

let collection = λf.λs.map f(s:int) on int;;
```

```
* ML sequent_manipulators

      letrec number_of_elts_in_list l =
        if l = [] then 0
        else 1 + number_of_elts_in_list (tl l)
      ;;

      letrec ith_elt_in_list i l =
        if l = [] then failwith `list too short`
        if i<1 then failwith `negative argument`
        if i=1 then hd l
        else ith_elt_in_list (i-1) (tl l)
      ;;
```

```
* ML finset_term_predicates

      let is_membership_term t =
        (destruct_term_of_theorem (fst (destruct_apply
            (fst (destruct_apply (fst (destruct_apply t))))))
        =
        `in_`)
        ?
```

```
  false
;;
```

* ML finset_term_destructors

```
let destruct_finset t =
  snd (destruct_apply t);;

%
Returns term#(term#term), with term1 being member,
term2 being
type, and term3 being set.
%
let destruct_membership t =
   ( snd (destruct_apply (fst (destruct_apply t))),
   ( snd (destruct_apply (fst (destruct_apply
                          (fst (destruct_apply t))))),
     snd (destruct_apply t)));;
```

* ML finset_equality_tactics

```
let FinsetEqualityIntro p =
  let terms, finset_type = destruct_equal (concl p) in
  let basetype = destruct_finset (finset_type) in
  let equand1 = hd terms in
  let equand2 = hd (tl terms) in
  (InstantiateLemma `extensionality`
                    [basetype;equand1;equand2]
  THENL
  [ Idtac
  ; Idtac
  ; Idtac
  ; Idtac
  ; \q. let l = number_of_elts_in_list (hypotheses q) in
       (Elim l
```

```
        THENL
        [ Idtac
        ; Thinning [l;l+1]
          THEN Elim l
          THEN Thinning [l;l+2]
          THEN BackThruHyp l
          THEN Intro
          THENW (Intro THEN Thinning [l])
        ]) q
  ]) p
;;

let FinsetEqualityElim i p =
  let equalityhyp = snd (destruct_declaration
                        (ith_elt_in_list i
                           (hypotheses p))) in
  let l = number_of_elts_in_list (hypotheses p) in
  let terms, finset_type = destruct_equal equalityhyp in
  let basetype = destruct_finset (finset_type) in
  let equand1 = hd terms in
  let equand2 = hd (tl terms) in
  let c = concl p in
  (InstantiateLemma `extensionality`
                    [basetype;equand1;equand2]
  THENL
  [ Idtac
  ; Idtac
  ; Idtac
  ; Idtac
  ; (if is_membership_term c then
      ref `explicit intro axiom` THEN Elim (l+1)
    else
      Elim (l+1)
    )
    THENL
    [ Idtac
    ; Thinning [l+1;l+2]
```

```
        THEN Elim (l+1)
        THEN Elim (l+3)
        THENL
        [ Idtac
        ; Thinning [i;l+1;l+2;l+3]
          THEN (if is_membership_term c
                then
                  let member,type,set=
                      destruct_membership c in
                  InstantiateLemma `in_intro_axiom_lemma`
                                   [type;set;member]
                  THENL
                  [ Idtac
                  ; Idtac
                  ; Idtac
                  ; Thinning [l+1]
                  ; Idtac
                  ; Idtac
                  ]
                else Idtac)
        ]
      ]
    ]) p
  ;;
```

```
* THM function_range_expansion

      >>∀S,T:U1.∀P:T->U1.∀f:S->{x:T|P(x)}.f in S->T


* THM finset_subtype_lemma

      >>∀T:U1.∀P:T->U1.∀s:finset({x:T|P(x)}).s in finset(T)


* DEF SynTrees
```

```
************************************
Synchronization Trees
**************************************==
```

```
* DEF action

     Action==
     int


* THM st_

     >>U1

     Extraction:
     rec(st.finset(Action#st))


* DEF st

     ST==
     term_of(st_)


* THM act_of_

     >>(Action#ST)->Action

     Extraction:
     λb.b.1


* DEF act_of

     action of (<b:branch>)==
```

```
    term_of(act_of_)(<b>)


* THM cont_of_

    >>(Action#ST)->ST

    Extraction:
    λb.b.2


* DEF cont_of

    continuation of (<b:branch>)==
    term_of(cont_of_)(<b>)


* THM apply_to_cont_

    >>(ST->ST)->(Action#ST)->(Action#ST)

    Extraction:
    λf.λb.<action of (b),f(continuation of (b))>


* DEF apply_to_cont

    <f:function> continuation==
    term_of(apply_to_cont_)(<f>)


* DEF alt_apply_to_cont

    <f:function> the continuation==
    λ_.<_.1,<f>(_.2)>
```

* THM alt_apply_to_cont_

```
>>∀T,U:U1.∀f:T->U.
f the continuation in (Action#T)->(Action#U)

Extraction:
\T.\U.\f.(\v0.(\v1.(\v2.axiom)(v1(f)))(v0(U)))
(term_of(alt_apply_to_cont_)(T))
```

* THM finset_to_ST

```
>>∀s:finset(Action#ST).s in ST
```

* THM ST_to_finset

```
>>∀s:ST.s in finset(Action#ST)
```

* THM continuation_subtype_lemma

```
>>∀P:ST->U1.∀t:finset(Action#{x:ST|P(x)}).
t in finset(Action#ST)
```

* ML st_tactics

```
let SynTreeIntro =
   --* Lemma `finset_to_ST` THEN Try(EqualityIntro)
;;

let SynTreeUnroll v =
   InstantiateLemma `ST_to_finset` [v]
;;
```

```
let STTypeWellFormed =
   Try(ref `cumulativity`) THEN
   Seq ['ST in U1'] THENL
   [Idtac; \p. TopLevelComputeHyp (last (hypotheses p)) p]

;;

let SetAutotactic =
Repeat (Trivial
ORELSE Member
ORELSE Arith)
;;

let STAutotactic =
Repeat (Trivial
ORELSE EqualityIntro
ORELSE Member
ORELSE Arith)
;;

letrec SynTreeListUnroll l p =
   (if l=[] then Idtac
   else (Seq [make_equal_term 'finset(Action#ST)'

                 [(hd l)]]
        THENL
        [ SynTreeUnroll (hd l)
        ; SynTreeListUnroll (tl l)
        ])) p
;;

let UnrollSTInEqualityConcl p =
   let terms = fst (destruct_equal (concl p)) in
   let sequand =
       make_equal_term 'finset(Action#ST)' terms in
   (Seq [sequand]
   THENL
```

```
   [ Idtac
   ; SubstFor sequand
     THENL
     [ Idtac
     ; Idtac
     ; EqualityIntro
       THENL
       [ Idtac
       ; SynTreeIntro
       ; Idtac
       ]
     ]
   ]) p
;;

let UnrollSTInEqualityHyp i p =
   let hypi = ith_elt_in_list i (hypotheses p) in
   let terms = fst (destruct_equal
                   (snd (destruct_declaration hypi))) in
   let sequand =
       make_equal_term 'finset(Action#ST)' terms in
   (Seq [sequand]
   THENL
   [ SubstFor (make_equal_term 'ST' terms)
     THENL
     [ Idtac
     ; Idtac
     ; EqualityIntro
       THENL
       [ Idtac
       ; \q. SynTreeUnroll
             (hd (fst (destruct_equal (concl q)))) q
       ; Idtac
       ]
     ]
   ; Idtac
   ]) p
```

```
;;
```

```
* ML st_term_destructors

      let set_term_in_membership_term t =
         snd (destruct_apply t);;

      let member_term_in_membership_term t =
         snd (destruct_apply (fst (destruct_apply t)));;

      let type_term_in_membership_term t =
         snd (destruct_apply
             (fst (destruct_apply
                  (fst (destruct_apply t)))));;

      let first_summand_in_plus_term t =
         snd (destruct_apply (fst (destruct_apply t)));;

      let second_summand_in_plus_term t =
         snd (destruct_apply t);;

      let make_membership_term t1 t2 t3 =
         let th = make_term_of_theorem_term `in_` in
            make_apply_term
               (make_apply_term
                   (make_apply_term th t1) t2) t3;;
```

```
* ML st_tactics_1

      let ShowMembershipInSummand i p =
         let g = concl p in
         let type = type_term_in_membership_term g in
         let sumset = set_term_in_membership_term g in
         let action,cont =
            destruct_pair (member_term_in_membership_term g) in
```

```
   let summand1 = first_summand_in_plus_term sumset in
   let summand2 = second_summand_in_plus_term sumset in
   if i = 1
   then InstantiateLemma `plus_lemma_1`
        [summand1;summand2;cont;action] p
   else InstantiateLemma `plus_lemma_2`
        [summand1;summand2;cont;action] p
;;
```

```
* THM nil_

      >>ST

      Extraction:
      ∅Action#ST


* DEF nil

      NIL==
      term_of(nil_)


* THM prepend_

      >>Action->ST->ST

      Extraction:
      λa.λp.{<a,p>}Action#ST


* DEF prepend

      <a:Action><p:ST>==
      term_of(prepend_)(<a>)(<p>)
```

* **THM plus_**

```
>>ST->ST->ST

Extraction:
λx.λy.x ∪Action#ST y
```

* **DEF plus**

```
<p:ST>+<q:ST>==
term_of(plus_)(<p>)(<q>)
```

* **THM merge_**

```
>>finset(ST)->ST

Extraction:
λs.∪Action#ST(s)
```

* **DEF merge**

```
∑<s:finset-of-sts>==
term_of(merge_)(<s>)
```

* **THM prune_predicate_**

```
>>Action->(Action#ST)->U1

Extraction:
λa.λx.(¬x.1=a in Action)
```

```
* DEF prune_predicate

      ¬action part=<a:Action>==
      term_of(prune_predicate_)(<a>)


* THM prune_predicate_decidable

      >>∀a:Action.¬action part=a is decidable on Action#ST


* THM prune_

      >>Action->ST->ST

      Extraction:
      λa.λs.rec_ind(s; prune,t.map
         λb.<b.1,prune(b.2)>(Action#ST:Action#ST) on
            {xεt|¬action part=a &
                  term_of(prune_predicate_decidable)(a)}Action#ST)


* DEF prune

      <P:SynTree>~ <a:Action>==
      term_of(prune_)(<a>)(<P>)


* THM restriction_

      >>Action->ST->ST

      Extraction:
      λa.λs.(s~ a)~ (-a)
```

```
* DEF restriction

      <t:SynTree>\\<a:Action>==
      term_of(restriction_)(<a>)(<t>)


* THM weed_

      >>Action->(Action#ST)->U1

      Extraction:
      λa.λb.a=b.1 in Action


* DEF weed

      action part = <a:Action>==
      term_of(weed_)(<a>)


* THM weed_decidable

      >>∀a:Action.action part = a is decidable on Action#ST


* THM composition_

      >>ST->ST->ST

      Extraction:
      λs.rec_ind(s; rolls,s1.
      λt.(rec_ind(t; rollt,t1.
      λz.map (λb.<b.1,rollt(b.2)(z)>) on t1+
      map (λb.<b.1,rolls(b.2)(t1)>) on z+
      ∑map (λc.map (λb.<0,rolls(b.2)(c.2)>)
              on
              {xϵ{xϵz|¬action part=0 &
```

```
                    term_of(prune_predicate_decidable)(0)}
               |action part = -c.1 &
                term_of(weed_decidable)(-c.1)})
          on t1
     )(s1))
     )
```

```
* DEF composition

     <p:SynTree>|<q:SynTree>==
     term_of(composition_)(<p>)(<q>)


* THM arrow_

     >>ST->Action->ST->U1

     Extraction:
     λs.λa.λt.<a,t> ϵAction#ST s


* DEF arrow

     <s:SynTree>--<a:Action>-→ <t:SynTree>==
     term_of(arrow_)(<s>)(<a>)(<t>)


* THM nil_semantics

     >>∀P:ST.∀a:Action.¬(NIL--a-→ P)


* THM prepend_semantics

     >>∀P:ST.∀a:Action.aP--a-→ P
```

* **THM** plus_semantics_1

  >>∀P,Q,P1:ST.∀a:Action.
  P--a-→ P1 => P+Q--a-→ P1

* **THM** plus_semantics_2

  >>∀P,Q,Q1:ST.∀a:Action.
  Q--a-→ Q1 => P+Q--a-→ Q1

* **THM** plus_lemma_1

  >>∀P,Q,P1:ST.∀a:Action.
  <a,P1> ϵAction#ST P => <a,P1> ϵAction#ST P+Q

* **THM** plus_lemma_2

  >>∀P,Q,Q1:ST.∀a:Action.
  <a,Q1> ϵAction#ST Q => <a,Q1> ϵAction#ST P+Q

* **THM** finset_of_ST_lemma

  >>∀s:finset(ST).s in finset(finset(Action#ST))

* **THM** in_ST_lemma

  >>∀s:finset(ST).∀P:ST.P ϵfinset(Action#ST) s => P ϵST s

* **THM** in_finset_of_ST_lemma

```
>>∀s:finset(ST).∀P:ST.P εST s => P εfinset(Action#ST) s
```

* THM merge_semantics

```
>>∀s:finset(ST).∀a:Action.∀P1:ST.
∃P:ST.P εST s & P--a-→ P1 => ∑s--a-→ P1
```

* THM merge_lemma

```
>>∀s:finset(ST).∀a:Action.∀P1:ST.
∃P:ST.P εST s & <a,P1> εAction#ST P =>
<a,P1> εAction#ST ∑s
```

* THM prune_semantics

```
>>∀P,P1:ST.∀a,b:Action.
a≠ b in Action => P--a-→ P1 => (P~ b)--a-→ (P1~ b)
```

* THM restriction_semantics

```
>>∀P,P1:ST.∀a,b:Action.
a≠ b in Action => a≠ -b in Action =>
P--a-→ P1 => P\b--a-→ P1\b
```

* THM composition_subterm_1_lemma

```
>>∀P,Q:ST.map (λb.<b.1,P|b.2>) on Q in ST
```

* THM composition_subterm_2_lemma

```
>>∀P,Q:ST.map (λb.<b.1,b.2|Q>) on P in ST
```

* THM composition_subterm_3_lemma

```
>>∀P,Q:ST.
map (λc.map (λb.<0,b.2|c.2>)
on {xϵ{xϵP|¬action part=0 &
           term_of(prune_predicate_decidable)(0)}
   |action part = -c.1 & term_of(weed_decidable)(-c.1)})
on Q in finset(ST)
```

* THM expansion_thm

```
>>∀P,Q:ST.
P|Q
=
map (λb.<b.1,P|b.2>) on Q+
map (λb.<b.1,b.2|Q>) on P+
∑map (λc.map (λb.<0,b.2|c.2>)
on {xϵ{xϵP|¬action part=0 &
          term_of(prune_predicate_decidable)(0)}
   |action part = -c.1 &
    term_of(weed_decidable)(-c.1)})
on Q
in ST
```

* THM composition_semantics_1

```
>>∀P,Q,P1:ST.∀a:Action.
P--a-→ P1 => P|Q--a-→ P1|Q
```

* THM composition_semantics_2

```
>>∀P,Q,Q1:ST.∀a:Action.
Q--a-→ Q1 => P|Q--a-→ P|Q1
```

* THM composition_semantics_3

```
>>∀P,Q,P1,Q1:ST.∀a:{x:Action|x≠ 0 in Action}.
P--a-→ P1 => Q--(-a)-→ Q1 => P|Q--0-→ P1|Q1
```

* THM equality_semantics_1

```
>>∀P,Q:ST.
(∀a:Action.
(∀P1:ST.P--a-→ P1 =>
    ∃Q1:ST.(Q--a-→ Q1 & P1=Q1 in ST))
&
(∀Q1:ST.Q--a-→ Q1 =>
   ∃P1:ST.(P--a-→ P1 & P1=Q1 in ST)))
=>
P=Q in ST
```

* THM equality_semantics_2

```
>>∀P,Q:ST.P=Q in ST =>
∀a:Action.
(∀P1:ST.P--a-→ P1 => ∃Q1:ST.(Q--a-→ Q1 & P1=Q1 in ST))
&
(∀Q1:ST.Q--a-→ Q1 => ∃P1:ST.(P--a-→ P1 & P1=Q1 in ST))
```

* ML CCS_equality_tactics

```
let CCSEqualityIntro p =
  let terms = fst (destruct_equal (concl p)) in
  let l = number_of_elts_in_list (hypotheses p) in
```

```
  let equand1 = hd terms in
  let equand2 = hd (tl terms) in
  (InstantiateLemma `equality_semantics_1` [equand1;equand2]

  THENL
  [ Idtac
  ; Idtac
  ; Thinning [l+1] THEN (RepeatFor 3 Intro)
  ; Idtac
  ; Idtac
  ]) p
;;
```

```
* DEF sem

      sem(<P:P-signal>,<V:V-signal>)==
      (-<P>)(-<V>)(-<P>)(-<V>)NIL
```

```
* EVAL test

      let t1 = 12NIL;;

      let t2 = 34NIL;;

      let sum = t1+t2;;

      let t3 = 5sum;;

      let t4 = 1NIL;;

      let t5 = -1NIL;;

      let traces =
      λs.rec_ind(s; lister,t.int_eq(t.1;0;
                   {nil}int list;
```

```
                  ∪int list(map λb.(map λl.(b.1.l)
                                   on lister(b.2)) on t)));;

      let list_set = λs.ind(s.1;u,v.nil;nil;u,v.(s.2(u).v));;

      let list_traces = λs.list_set(traces(s));;


* EVAL mutex_defs

      let P = 1;;

      let V = 2;;

      let a = 3;;

      let b = 4;;

      let c = 5;;

      let d = 6;;

      let p1 = PabVNIL;;

      let p2 = PcdVNIL;;

      let sem = -P-V-P-VNIL;;


* EVAL mutex_eval

      let l = list_traces (p1|sem~ P~ V|p2\P\V);;
```

Bibliography

[Abramsky 83]
Abramsky, S.
"On Semantic Foundations for Applicative Multiprogramming."
Proceedings of the 10th Annual Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 154, J. Diaz ed., pp. 1-14.
Springer-Verlag, Berlin, 1984.

[Aczel 85]
Aczel, P.
"Lecture Notes on Non-Well-Founded Sets."
Lecture notes for a course at Stanford University, March, 1985.

[Allen 87a]
Allen, S.
"A Non-Type-Theoretic Account of Martin-Lof's Types."
To appear in the *Proceedings of the 1987 Symposium on Logic in Computer Science.*

[Allen 87b]
Allen, S.
Cornell University Ph.D. Thesis, to appear 1987.

[Alpern and Schneider 86]
Alpern, B. and F. Schneider.
"Recognizing Safety and Liveness."
Technical Report 86-727, Department of Computer Science, Cornell University, January 1986.

[Apt, Francez and deRoever 80]
Apt, K., N. Francez and W.-P. deRoever.
"A Proof System for Communicating Sequential Processes."
Association for Computing Machinery Transactions on Programming Languages and Systems 2, n. 3, July 1980, pp.359-385.

[Arvind, Kathail and Pingali 85]
Arvind, V. Kathail and K. Pingali.
"Sharing of Computation in Functional Language Implementations."
Proceedings of the Workshop on Implementation of Functional Languages, L. Augustsson, J. Hughes, T. Johnsson and K.Karlsson eds., pp..
Programming Methodology Group Report 17, University of Goteborg and Chalmers University of Technology, February 1985.

[Bates and Constable 85]
Bates, J. and R. Constable.
"Proofs as Programs."
Association for Computing Machinery Transactions on Programming Languages and Systems 7, n. 1, January 1985, pp.113-136.

[Beck and Cleaveland 87]
Beck, M. and R. Cleaveland.
"Making Use of Knowledge in Programming."
To appear as Cornell technical report.

[Bishop 67]
Bishop, E.
Foundations of Constructive Analysis.
McGraw-Hill, New York, 1967.

[Brookes 83]
Brookes, S.
"On the Relationship of CCS and CSP."
Technical Report CMU-CS-83-111, Department of Computer Science, Carnegie-Mellon University, March 1983.

[Brookes 85]
Brookes, S.
"An Axiomatic Treatment of a Parallel Programming Language."
Proceedings of Logics of Programs 1985, Lecture Notes in Computer Science 198, R. Parikh ed., pp.44-60.
Springer-Verlag, Berlin, 1985.

[Brookes 86]
Brookes, S.
"A Semantically Based Proof System for Partial Correctness and Deadlock in CSP."
Proceedings of the 1986 Symposium on Logic in Computer Science, pp. 58-65.
IEEE Computer Society Press, Washington, DC, 1986.

[Brookes, Hoare and Roscoe 84]
Brookes, S., C.A.R. Hoare and A. Roscoe.
"A Theory of Communicating Sequential Processes."
Journal of the Association for Computing Machinery 31, n. 3, July 1984, pp. 560-599.

[Brookes and Roscoe]
Brookes, S. and A. Roscoe.
"An Improved Failures Model for Communicating Processes."
Technical Report, Department of Computer Science, Carnegie-Mellon University.

[Brouwer 23]
Brouwer, L.E.J.
"On the Significance of the Principle of Excluded Middle in Mathematics, Especially in Function Theory."
In *From Frege to Godel: A Sourcebook in Mathematical Logic*, J. vanHeijenoort ed., pp. 334-345.
Harvard University Press, Cambridge, MA, 1967.

[deBruijn 80]
deBruijn, N.
"A Survey of the Project AUTOMATH."
To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism, J. Seldin and J. Hindley eds., pp. 589-606.
Academic Press, New York, 1980.

[Chandy and Misra 1985]
Chandy, K. and J. Misra.
"How Processes Learn."
Proceedings of the 4th Association for Computing Machinery Symposium on the Principles of Distributed Computing, pp. 204-214.
The Association for Computing Machinery, New York, 1985.

[Clack and Peyton Jones 85]
Clack, C. and S. Peyton Jones.
"Strictness Analysis--A Practical Approach."
Proceedings of EUROCAL '85, v. 1, Lecture Notes in Computer Science 203, B. Buchberger ed., pp. 35-49.
Springer-Verlag, Berlin, 1985.

[Cleaveland and Panangaden 85]
Cleaveland, R. and P. Panangaden.
"Type Theory and Concurrency."
Technical Report 85-714, Department of Computer Science, Cornell University.

[Constable and Mendler 85]
Constable, R. and N. Mendler.
"Recursive Definitions in Type Theory."
Proceedings of Logics of Programs 1985, Lecture Notes in Computer Science 193, R. Parikh ed., pp. 61-78.
Springer-Verlag, Berlin, 1985.

[Constable et al. 86]
Constable, R. et al.
Implementing Mathematics with the Nuprl Proof Development System.
Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Coquand and Huet 85]
Coquand, T. and G. Huet.
"Constructions: A Higher-Order Proof System for Mechanizing Mathematics."
Proceedings of EUROCAL '85, v. 1, Lecture Notes in Computer Science 203, B. Buchberger ed., pp. 35-49.
Springer-Verlag, Berlin, 1985.

[Curry, Feys and Craig 58]
Curry, H., R. Feys and W. Craig.
Combinatory Logic, v. 1.
North-Holland, Amsterdam, 1958.

[Gordon, Milner and Wadsworth 79]
Gordon, M., A. Milner and C. Wadsworth.
Edinburgh LCF: A Mechanized Logic of Computation.
Lecture Notes in Computer Science 78.
Springer-Verlag, Berlin, 1979.

[Gries 81]
Gries, D.
The Science of Programming.
Springer-Verlag, New York, 1981.

[Griffin 87]
Griffin, T.
"An Environment for Formal Systems."
To appear as a technical report, Department of Computer Science, Cornell University, June 1987.

[Hailpern 82]
Hailpern, B.
Verifying Concurrent Processes Using Temporal Logic.
Lecture Notes in Computer Science 129.
Springer-Verlag, Berlin, 1982.

[Halpern and Moses 84]
Halpern, J. and Y. Moses.
"Knowledge and Common Knowledge in a Distributed Environment."
Proceedings of the 3rd Association for Computing Machinery Symposium on Principles of Distributed Computing, 1984, pp. 50-61.
Revised as IBM Research Report RJ 4421.

[Harper, Honsell and Plotkin 87]
Harper, R., F. Honsell and G. Plotkin.
"A Framework for Defining Logics."
To appear in the *Proceedings of the 1987 Symposium on Logic in Computer Science.*

[Hennessy 82]
Hennessy, M.
"Synchronous and Asynchronous Experiments on Processes."
Technical Report CSR-125-82, Department of Computer Science, University of Edinburgh, September 1982.

[Hennessy 83]
Hennessy, M.
"A Model for Nondeterministic Machines."
Technical Report CSR-135-83, Department of Computer Science, University of Edinburgh, June 1983.

[Hennessy and Milner 85]
Hennessy, M. and R. Milner.
"Algebraic Laws for Nondeterminism and Concurrency."
Journal of the Association for Computing Machinery 32, n. 1, January 1985, pp.137-161.

[Heyting 30]
Heyting, A.
"Die formalen Regeln der intuitionistichen Logik."
Sigzundsber. Preuss. Akad. Wiss., Phy.s-Math Kl., 1930, pp. 42-56.

[Hoare 78]
Hoare, C.A.R.
"Communicating Sequential Processes."
Communications of the Association for Computing Machinery 21, n. 8, August 1978, pp. 666-677.

[Hoare 85]
Hoare, C.A.R.
Communicating Sequential Processes.
Prentice-Hall International, London, 1985.

[Hooman and deRoever 86]
Hooman, J. and W.-P. deRoever.
"The Quest Goes on: A Survey of Proofsystems for Partial Correctness of CSP."
Current Trends in Concurrency, Lecture Notes in Computer Science 224, J. deBakker, W.-P. deRoever and G. Rozenberg eds., pp. 343-395.
Springer-Verlag, Berlin, 1986.

[Howard 80]
Howard, W.
"The Formulas-as-Types Notion of Construction."
To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, J. Seldin and J. Hindley eds., pp. 479-490.
Academic Press, New York, 1980.

[Howe 87]
Howe, D.
"The Computational Behaviour of Girard's Paradox."
To appear in the *Proceedings of the 1987 Symposium on Logic in Computer Science.*

[Hudak and Young 86]
"Higher-Order Strictness Analysis in Untyped Lambda Calculus."
Proceedings of the 13th Annual Association for Computing Machinery Conference on the Principles of Programming Languages, pp. 97-109.
The Association for Computing Machinery, New York, 1986.

[Hughes and Cresswell 68]
Hughes, G. and M. Cresswell.
An Introduction to Modal Logic.
Methuen, London, 1968.

[Keller and Panangaden 84]
Keller, R. and P. Panangaden.
"Semantics of Networks Containing Indeterminate Operators."
Seminar on Concurrency, Lecture Notes in Computer Science 197, S. Brooks, A. Roscoe and G. Winskel eds., pp. 479-496.
Springer-Verlag, Berlin, 1984.

[Knoblock 87]
Knoblock, T.
Cornell University Ph.D. Thesis, to appear 1987.

[Knoblock and Constable 86]
Knoblock, T. and R. Constable.
"Formalized Metareasoning in Type Theory."
Proceedings of the 1986 Symposium on Logic in Computer Science, pp. 237-248.
IEEE Computer Society Press, Washington, DC, 1986.

[Lamport 77]
Lamport, L.
"Proving the Correctness of Multiprocess Programs."
IEEE Transactions on Software Engineering SE-3, n. 2, March 1977, pp. 125-143.

[Lamport and Schneider 82]
Lamport, L. and F. Schneider.
"The 'Hoare Logic' of CSP, and All That."
Technical Report 82-490, Department of Computer Science, Cornell University, May 1982.

[Levin and Gries 82]
Levin, G. and D. Gries.
"A Proof Technique for Communicating Sequential Processes."
Acta Informatica 15, n.3, June 1981, pp. 281-302.

[Manna and Pnueli 82]
Manna, Z. and A. Pnueli.
"Verification of Concurrent Programs: Temporal Proof Principles."
Proceedings of Logics of Programs 1981, Lecture Notes in Computer Science 131, D. Kozen ed., pp. 200-252.
Springer-Verlag, Berlin, 1982.

[Martin-Lof 71]
Martin-Lof, P.
"A Theory of Types."
Technical Report 71-3, Department of Mathematics, University of Stockholm, February 1971.

[Martin-Lof 82]
Martin-Lof, P.
"Constructive Mathematics and Computer Programming."
Proceedings of the Sixth Internation Congress for Logic, Methodology and Philosopy of Science, pp. 153-175.
North-Holland, Amsterdam, 1982.

[McCarty 84]
McCarty, D.
Realizability and Recursive Mathematics.
Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, 1984.

[Mendler, Panangaden and Constable 86]
Mendler, N., P. Panangaden and R. Constable.
"Infinite Objects in Type Theory."
Proceedings of the 1986 Symposium on Logic in Computer Science, pp. 249-257.
IEEE Computer Society Press, Washington, DC, 1986.

[Milne 85]
Milne, G.
"CIRCAL and the Representation of Communication, Concurrency and Time."
Association for Computing Machinery Transactions of Programming Languages 7, n. 2, April 1985, pp. 270-298.

[Milner 80]
Milner, R.
A Calculus of Communicating Systems.
Lecture Notes in Computer Science 92.
Springer-Verlag, Berlin, 1980.

[Milner 83]
Milner, R.
"Calculi for Synchrony and Asynchrony."
Theoretical Computer Science 25, n. 3, July 1983, pp. 267-310.

[Milner 86]
Milner, R.
"A Complete Axiomatisation of Observational Congruence of Finite-State Behaviours."
LFCS Report ECS-LFCS-86-8, Department of Computer Science, University of Edinburgh, August 1986.

[Misra and Chandy 81]
Misra, J. and K. Chandy.
"Proofs of Networks of Processes."
IEEE Transactions on Software Engineering SE-7, n. 4, July 1981, pp. 417-426.

[Mycroft 80]
Mycroft, A.
"The Theory and Practice of Transforming Call-by-Need into Call-by-Value."
Proceedings of the 1980 International Symposium of Programming, Lecture Notes in Computer Science 83, B. Robinet ed., pp. 269-281.
Springer-Verlag, Berlin, 1985.

[Nguyen, Demers, Gries and Owicki 85]
Nguyen, V., A. Demers, D. Gries and S. Owicki.
"A Model and Temporal Proof System for Networks of Processes."
Technical Report 85-690, Department of Computer Science, Cornell University, June 1985.

[deNicola and Hennessy 82]
deNicola, R. and M. Hennessy.
"Testing Equivalences for Processes."
Technical Report CSR-123-82, Department of Computer Science, University of Edinburgh, August 1982.

[Owicki and Gries 76]
Owicki, S. and D. Gries.
"An Axiomatic Proof Technique for Parallel Programs."
Acta Informatica 6, 1976, pp. 319-340.

[Owicki and Lamport 82]
Owicki, S. and L. Lamport.
"Proving Liveness Properties of Concurrent Programs."
Association for Computing Machinery Transactions in Programming Languages and Systems 4, n. 3, July 1982, pp. 455-495.

[Peleg 84]
Peleg, D.
"Concurrent Dynamic Logic."
Technical Report CS84-14, Department of Applied Mathematics, the Weizmann Institute of Science, July 1984.

[Plotkin 76]
Plotkin, G.
"A Powerdomain Construction."
SIAM Journal of Computing 5, n. 3, September 1976, pp. 452-487.

[Plotkin 82]
Plotkin, G.
"An Operational Semantics of CSP."
Technical Report CSR-114-82, Department of Computer Science, University of Edinburgh, May 1982.

[Pnueli 77]
Pnueli, A.
"The Temporal Logic of Programs."
Proceedings of the 18th Annual Symposium on the Foundations of Computer Science, 1977, pp. 46-57.
IEEE Computer Society Press, Washington, DC, 1977.

[Pnueli 79]
Pnueli, A.
"The Temporal Semantics of Concurrent Programs."
Proceedings of the International Symposium on the Semantics of Concurrent Computation, Lecture Notes in Computer Science 70, G. Rahn ed., pp. 1-20.
Springer-Verlag, Berlin, 1979.

[Pnueli 86]
Pnueli, A.
"Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends."
Current Trends in Concurrency, Lecture Notes in Computer Science 224, J. deBakker, W.-P. deRoever and G. Gozenberg eds., pp. 510-584.
Springer-Verlag, Berlin, 1986.

[Scott 76]
Scott, D.
"Data Types as Lattices."
SIAM Journal of Computing 5, n. 3, September 1976, pp. 522-587.

[Shoenfeld 67]
Shoenfeld, J.
Mathematical Logic.
Addison-Wesley, Reading, MA, 1967.

[Smyth 78]
Smyth, M.
"Power Domains."
Journal of Computer and System Sciences 16, n. 1, February 1978, pp. 23-36.

[Soundararajan and Dahl 82]
Soundararajan, N. and O. Dahl.
"Partial Correctness Semantics for Communicating Sequential Processes."
Research Report 66, Institute for Informatics, University of Oslo, 1982.

[Stirling 85a]
Stirling, C.
"A Complete Modal Proof System for a Subset of SCCS."
Proceedings of the 1985 Colloquium on Trees in Algebra and Programming, Lecture Notes in Computer Science 185, H. Ehrig, C.Floyd, M. Nivat and J. Thatcher eds., pp. 253-266.
Springer-Verlag, Berlin, 1985.

[Stirling 85b]
Stirling, C.
"Modal Logics for Communicating Systems."
Technical Report CSR-193-85, Department of Computer Science, University of Edinburgh, October 1985.

[Stoy 77]
Stoy, J.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
MIT Press, Cambridge, MA, 1977.

[Strom 86]
Strom, R.
Private communication, May 1986.

[Teitelbaum and Reps 81]
Teitelbaum, R. and T. Reps.
"The Cornell Program Synthesizer: A Syntax-Directed Programming Environment."
Communications of the Association for Computing Machinery 24, n. 9, September 1981, pp. 563-573.

[Widom, Gries and Schneider 86]
Widom, J., D. Gries and F. Schneider.
"Completeness and Incompleteness of Trace-Based Network Proof Systems."
Technical Report 86-766, Department of Computer Science, Cornell University, July 1986.

[Winskel 83a]
Winskel, G.
"Synchronization Trees."
Proceedings of the 10th Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 154, J. Diaz ed., pp. 695-711.
Springer-Verlag, Berlin, 1982.

[Winskel 83b]
Winskel, G.
"A Note on Powerdomains and Modality."
Technical Report CMU-CS-83-159, Department of Computer Science, Carnegie-Mellon University, September 1983.

[Winskel 84]
Winskel, G.
"On the Composition and Decomposition of Assertions."
Seminar on Concurrency, Lecture Notes in Computer Science 197, S. Brookes, A. Roscoe and G. Winskel eds., pp. 62-75.
Springer-Verlag, Berlin, 1984.

[Winskel 86]
Winskel, G.
"A Complete Proof System for SCCS with Modal Assertions."
Proceedings of the Fifth Conference on the Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 206, S. Maheshwari ed., pp. 392-410.
Springer-Verlag, Berlin, 1986.