Type Theory and Concurrency

Rance Cleaveland* Prakash Panangaden†

May 13, 1996

Abstract

This paper describes the use of an automated reasoning tool, the Nuprl system, to formalize Milner's Calculus of Communicating Systems (CCS). The goals of this work are two-fold: the first is to investigate the feasibility of using systems like Nuprl to handle the formal detail arising from reasoning about concurrency, while the second is to develop a framework in which various formalisms for reasoning about concurrency may be presented in an automated fashion. To these ends, an implementation in Nuprl of a formal theory of concurrency is described, an implementation of CCS in this mechanized semantic theory presented, and two means of analyzing CCS terms are investigated.

1 Introduction

Many formalisms for reasoning about concurrent programming have been developed in the past decade, as interest in parallel computation has waxed considerably. In addition to various Hoare-style logics, which are generally used to reason about programs written in sequential imperative languages with concurrent programming constructs added, modal logics, in particular temporal logic, have been extensively explored as a means for specifying and verifying programs. Furthermore, algebraic formalisms have been proposed that define equational theories for reasoning about concurrent programs. These three styles of formalism embody very different techniques, but one characteristic that they all share is the rapid increase in complexity of verification obligations as the size of the system to be verified increases. Indeed, the level of complexity involved in understanding parallel programs seems inevitable, given the number of ways in which processes may interact with each other.

*Author's address: Computer Science Department, Mathematics and Physical Sciences Building, University of Sussex, Falmer, Brighton BN1 9QH, ENGLAND.

†Author's address: Computer Science Department, Upson Hall, Cornell University, Ithaca, New York 14853, USA.

Another area of mathematics, formal logic, also involves the handling of large amounts of detail. Formal mathematical arguments have the advantage of more often being correct than informal ones, due to the precision that formalism forces on the mathematician, but the amount of formal detail arising from even the most elementary of informal arguments has traditionally made formal mathematics unusable in practice. However, recent research into the automation of formal theories suggests that at least some of the detail can be automatically handled. A natural question that arises, then, involves the practicability of using systems like these to reason about concurrency, and this paper addresses this question by describing an application of the Nuprl proof development system to the development of a formal theory of communicating processes. More specifically, this paper presents a semantic theory of nondeterminism implemented in Nuprl and an implementation of the Calculus for Communicating Systems (CCS) of Milner in Nuprl using the theory. As Nuprl can be used to model various kinds of predicate calculi, it can be used in conjunction with the aforementioned semantic theory to model the deductive schemes of various modal logics; one such representative logic, Stirling's axiomatization of a fragment of the Hennessy-Milner logic [18], is presented. Process constructors from various other algebraic formalisms, including those of the Synchronous Calculus of Communicating Systems (SCCS) due to Milner [17] and the Communicating Sequential Processes (CSP) theory of Brookes, Hoare and Roscoe [2, 11], may also be described.

The motivation for the work in this paper has three main sources. The first has already been mentioned, and that is that by formalizing a theory of concurrency in Nuprl the facilities supplied by the Nuprl system should enable much of the detail arising from formal arguments of the correctness of concurrent processes to be handled automatically. The second motivation stems from a desire to present an automated framework that underlies different formalisms for reasoning about concurrency. The semantic theory we implement in Nuprl is expressive enough to give accounts of the operators of CCS, of a version of SCCS (one with a finitary version of sum) and of CSP. The semantic basis of our approach also enables us to develop logics concerned with the meanings of programs, rather than the programs themselves; this means that the same logic developed in this framework can be used to reason about processes specified in any language whose semantics can be accounted for in our framework. The final motivation is one of consistency. The Nuprl theory is consistent; hence, any theory modeled in it will likewise be consistent. Consistency can be a nontrivial issue in formalisms with many axioms and rules of inference, and as many theories of concurrency are complicated, the consistency ensured by implementing them in a system like Nuprl is desirable. It is also the case that the Nuprl system interactively checks proofs as they are carried out; hence, any completed proof of the correctness of a system in our framework is guaranteed to itself be correct.

The rest of the paper is organized along the following lines. Section 2 describes CCS. Section 3 describes the semantic theory and contains a development of CCS in the theory. Section 4 presents an overview of the Nuprl logic and system, and section 5 describes our implementation of the semantic theory in Nuprl and our implementation of CCS. Section 6 presents a small example involving the use of Nuprl to argue for the correctness of a binary semaphore and shows how modal logics may be implemented in our framework. The final section contains conclusions and directions for further research.

2 CCS

CCS [16] is an algebraic formalism for reasoning about concurrency. Other algebraic formalisms, including SCCS [17] and CSP [11], have been proposed; these will be discussed at the end of the section.

2.1 Syntax and Semantics

CCS processes are described on the basis of the atomic actions they perform during the course of their execution. Given a set of atomic actions, the calculus specifies certain operators that allow the construction of processes from actions and other processes. CCS represents concurrency in terms of nondeterminism; parallel execution is modeled as the interleaving of the actions of processes running in parallel. It is also possible for actions within processes to synchronize with certain actions in other processes.

In order to describe the CCS process constructors formally we first describe the structure of the set A of atomic actions. A must meet the following conditions.

1. There is a distinguished element $\tau \in A$.
2. For every $a \in A$ with $a \neq \tau$ there is an element $\overline{a} \in A$, with $\overline{\overline{a}} = a$.

The action τ is called the *silent* action, while actions a and $\overline{a}$ are called *inverses*. Intuitively, τ represents an internal action performed by a process, while a and $\overline{a}$ represent actions that may synchronize with one another.

Certain functions from action sets to action sets, called *relabelings*, are used in what follows. Given action sets A and B, a function L from A to B is a relabeling if

1. $L(\tau) = \tau$, and
2. if $a \neq \tau$ then $L(\overline{a}) = \overline{L(a)}$.

$$
\begin{array}{rcll}
t & ::= & nil & \\
 & \mid & v & \text{for } v \in V \\
 & \mid & at & \text{for } a \in A \\
 & \mid & t + t & \\
 & \mid & t|t & \\
 & \mid & t\backslash a & \text{for } a \in A \\
 & \mid & t[L] & \text{for } L \text{ a relabeling} \\
 & \mid & \mathit{fix}(v.t) & \text{for } v \in V
\end{array}
$$

Figure 1: The syntax of CCS terms.

Given a set A satisfying the above conditions, and a set of variables V, figure 1 describes the syntax of CCS terms. Additionally, there is a syntactic constraint on t in $\mathit{fix}(v.t)$ that states that v must be *guarded* in t. A variable v is guarded in an expression t if every occurrence of v falls with the scope of a subterm of t having the form at' for some action a. In what follows the closed terms are of particular interest; we call these *processes*. P, Q and R will be used to range over processes.

CCS terms have the following intuitive semantics. The process denoted by nil is a null process; it can perform no actions and terminates immediately. The term at performs an a action and afterward behaves like t. Term $t + u$ behaves like either t or u, with the initial actions of both t and u initially available. The term $t|u$ represents the (interleaved) concurrent execution of t and u with possible synchronizations of inverse actions. $t\backslash a$ behaves like t except that executions beginning with action a or with action $\overline{a}$ are disallowed. If L is a relabeling then $t[L]$ represents the obviously relabeled term. Finally, $\mathit{fix}(v.t)$ is a recursively defined process representing the "solution" to the equation

$$v = t.$$

Since v is guarded in t this equation has a unique solution with respect to a behavioral equivalence defined in the next section [17]. The fix term constructor implicitly allows the definition of nonterminating processes; for example, the process $\mathit{fix}(v.av)$ successively performs a actions without ever terminating.

These intuitive semantics can be captured in terms of *labeled transition systems*. Formally, a labeled transition system T is a triple $\langle S, L, \{ R_l \mid l \in L \} \rangle$, where S is a set of states, L is a set of labels, and each R_l is a binary relation on states called a *derivation relation*. Intuitively, $s_1 R_l s_2$ specifies that s_1 may perform l and evolve to s_2. When $s_1 R_l s_2$ we shall write $s_1 \xrightarrow{l} s_2$.

In order to give the operational semantics of CCS as a labeled transition system, we identify the action set A as the set of labels and the set of terms as the set of states. The derivation

$$
\begin{array}{rcl}
at \xrightarrow{a} t & & \\
t \xrightarrow{a} t' & \Rightarrow & t + u \xrightarrow{a} t' \\
u \xrightarrow{b} u' & \Rightarrow & t + u \xrightarrow{b} u' \\
t \xrightarrow{a} t' & \Rightarrow & t|u \xrightarrow{a} t'|u \\
u \xrightarrow{b} u' & \Rightarrow & t|u \xrightarrow{b} t|u' \\
t \xrightarrow{a} t', u \xrightarrow{\overline{a}} u' & \Rightarrow & t|u \xrightarrow{\tau} t'|u' \\
t \xrightarrow{a} t', a \notin \{b, \overline{b}\} & \Rightarrow & t \backslash b \xrightarrow{a} t' \backslash b \\
t \xrightarrow{a} t' & \Rightarrow & t[L] \xrightarrow{L(a)} t'[L] \\
t[\mathit{fix}(v.t)/v] \xrightarrow{a} t' & \Rightarrow & \mathit{fix}(v.t) \xrightarrow{a} t'
\end{array}
$$

Figure 2: The semantics of CCS terms.

relations are defined in figure 2.1.

A theorem proven in [16] establishes a relationship between the $|$ and $+$ process constructors. Let I and J be finite sets of indices, and let $\sum_I$ represent the intuitive generalization of $+$. The theorem, called the *expansion theorem*, is stated as follows.

Theorem 2.1 *Let* $P = \sum_{i \in I} a_i P_i$ *and* $Q = \sum_{j \in J} b_j P_j$. *Then*

$$P|Q = \sum_{i \in I} a_i(P_i|Q) + \sum_{j \in J} b_j(P|Q_j) + \sum_{a_i = \overline{b_j}} \tau(P_i|Q_j).$$

Synchronization trees represent another means of describing the behavior of processes in CCS. A synchronization tree is a finitely branching, potentially infinite-depth tree with arcs labeled by actions. A path through a synchronization tree corresponds to a sequence of actions a process may perform, and the existence of two arcs emerging from a single node indicates the possibility of choice. The CCS process constructors mentioned previously correspond in a straightforward fashion to functions on synchronization trees.

2.2 Equivalences

Reasoning about processes in CCS can be done equationally on the basis of equivalences defined between processes. Milner presents three equivalences of interest in [16], and other equivalences have been proposed [2, 8, 7]. We shall concentrate on the former. The first equivalence, *strong equivalence*, can be defined in terms of binary relations on processes called *bisimulations* [17]. Intuitively, a bisimulation may be thought of as a two-place invariant on the states processes may enter. Formally, a bisimulation is a relation R such that PRQ implies that the following conditions hold.

1. If $P \xrightarrow{a} P'$ then there is a Q' such that $Q \xrightarrow{a} Q'$ and $P'RQ'$.

2. If $Q \xrightarrow{a} Q'$ then there is a P' such that $P \xrightarrow{a} P'$ and $P'RQ'$.

There is a weakest bisimulation that is the union of all bisimulations, and strong equivalence is defined as this bisimulation. We write $P{\sim}Q$ when P and Q are strongly equivalent.

Observational equivalence is an equivalence relation coarser than strong equivalence that attempts to capture the intuition that τ actions, being internal, ought not to enable processes to be distinguished. To capture this intuition formally, we first define an *observable derivation relation*, $\stackrel{a}{\Rightarrow}$ for $a \in A$, as follows. For a binary relation R, let R^* denote the reflexive and transitive closure of R. Then,

$$P \stackrel{a}{\Rightarrow} P' \text{ if and only if there exist } P_1, P_2 \text{ such that } P(\xrightarrow{\tau})^* P_1, P_1 \xrightarrow{a} P_2, \text{and} P_2(\xrightarrow{\tau})^* P'.$$

A *weak bisimulation* is a binary relation R on processes such that PRQ implies the following.

1. If $P \stackrel{a}{\Rightarrow} P'$ then either $a = \tau$ and $P'RQ$ or there is a Q' such that $Q \stackrel{a}{\Rightarrow} Q'$ and $P'RQ'$.

2. If $Q \stackrel{a}{\Rightarrow} Q'$ then either $a = \tau$ and PRQ' or there is a P' such that $P \stackrel{a}{\Rightarrow} P'$ and $P'RQ'$.

The union of all weak bisimulations is itself a weak bisimulation and is called observational equivalence. We write $P{\approx}Q$ when P and Q are observationally equivalent.

Some observations about these equivalences are in order. First, strong equivalence is a congruence, meaning that equivalent terms may be substituted for each other in any context; observational equivalence, on the other hand, is not. Accordingly, *observational congruence* is defined as the largest congruence contained within observational equivalence. Secondly, strong equivalence, observational equivalence and observational congruence have complete axiomatizations for finite processes; the interested reader is referred to [10].

2.3 SCCS and CSP

SCCS [17] is a variant of CCS that features a *synchronous* notion of parallel execution as opposed to the asynchronous, interleaved notion in CCS. In SCCS the action set A is an Abelian group, with the unit representing internal action and the group operation representing simultaneous execution. Formally, let A be an Abelian group with unit 1 and group operator $\circ$, and let V be a set of variables. Figure 3 describes the syntax of SCCS, while figure 4 summarizes its semantics.

There is a version of the expansion theorem for the SCCS composition operator. It is stated as follows.

$$
\begin{array}{rcll}
t & ::= & at & \text{for } a \in A \\
 & | & \sum\limits_{i \in I} t_i & \text{for } I \text{ an index set, } t_i \text{ a family of terms} \\
 & | & t \otimes t & \\
 & | & t\lceil H & \text{for } H \subseteq A,\, 1 \in H \\
 & | & \mathit{fix}(v.t) &
\end{array}
$$

Figure 3: SCCS syntax.

$$
\begin{array}{rcl}
 & & aP \xrightarrow{a} P \\
P_j \xrightarrow{a} P', \text{some} j \in I & \Rightarrow & \sum\limits_{i \in I} P_i \xrightarrow{a} P' \\
P \xrightarrow{a} P', Q \xrightarrow{b} Q' & \Rightarrow & P \otimes Q \xrightarrow{a \circ b} P' \otimes Q' \\
P \xrightarrow{a} P', a \in H & \Rightarrow & P\lceil H \xrightarrow{a} P'\lceil H \\
P[\mathit{fix}(x.P)/x] \xrightarrow{a} P' & \Rightarrow & \mathit{fix}(x.P) \xrightarrow{a} P'
\end{array}
$$

Figure 4: SCCS semantics.

Theorem 2.2 *Let* $P = \sum\limits_{i \in I} a_i P_i$ *and* $Q = \sum\limits_{j \in J} b_j P_j$. *Then*

$$P \otimes Q = \sum_{i \in I, j \in J} (a_i \circ b_j)(P_i \otimes Q_j).$$

Another algebraic language for describing processes, CSP [11], has operators that are slightly different from those of CCS and SCCS, although they are similar in spirit. The most notable departure involves the $\oplus$ operator. Intuitively, $P \oplus Q$ behaves like P or Q; one or the other is chosed nondeterministically. It is different from $P + Q$ in that $P \oplus Q$ may refuse to perform to the initial actions P or Q, depending on the outcome of the nondeterministic choice.

Several alternative equivalences also appear in the literature. Notable among these are testing equivalence [7, 8] and failures equivalence [2]. All of these equivalences have "trace-based" characterizations in the sense that they describe the behavior of processes on the basis of the characteristics of the states a process can be in after executing a "trace", or a finite sequence of actions. All these equivalences are coarser than observational equivalence.

3 A Denotational Model for Concurrency

This section develops a denotational model of concurrency inspired by Aczel [1]. This semantics serves as the basis of the implementation that is described later in the paper and is motivated by the following considerations. On the one hand, reasoning about meanings as opposed to programs is desirable, since with a general enough semantics one can reason about different programming languages using the same logic. It is also desirable that the equivalence in the denotational model reflects one of the operational equivalences described previously; moreover, the equivalence should also be a congruence, for then various syntactic manipulations (such as substitution) can be naturally modeled as semantic operations (such as function application, in the case of substitution). Since strong equivalence is a congruence, it is therefore a reasonable candidate for equality in a denotational model.

3.1 A Set-Theoretic Account of the Model

One of the equational laws satisfied by strong equivalence is the *absorption law*, which states that for any process P,

$$(P + P) \sim P.$$

The $+$ operator is also commutative and associative with respect $\sim$; it is suggestive to note that the set-theoretic notion of union has the same properties with respect to equality. The restriction operator removes some of the actions available to a process; analogously, the comprehension axiom of set theory allows the removal of elements from a set not satisfying a particular property. Moreover, relabeling alters every piece of a process in a uniform way, just as the collection axiom of set theory allows a set to be formed by applying a function to every element of another set. This suggests that a recursively defined set-theoretic model might have both the correct equivalence for our purposes as well as straightforward interpretations of the CCS process constructors. Intuitively, one might think of the set of processes ST as the solution to an equation of the form

$$ST = \mathit{finset}(A \times ST),$$

where A is the set of actions and $\mathit{finset}(X)$ denotes the collection of finite sets of set X. An element of ST consists of a finite set of pairs, the first component of which is an action and the second component of which is the "continuation" associated with the action.

The straightforwardness of this model is appealing, but a certain formal difficulty complicates its presentation somewhat. In traditional set theory, the existence of infinitely nested sets is proscribed by the *regularity axiom.* However, modeling nonterminating processes, such as $\mathit{fix}(v.av)$, in the scheme just described requires such infinitely nested, or *non-well-founded,* sets. At first glance this is a serious problem, since the regularity axiom excludes sets that give rise to paradoxes in set theories without this axiom. Aczel, however, solves this problem by altering the axioms of set theory to allow certain non-well-founded sets while still maintaining the consistency of set theory. Accordingly, the next section describes briefly Aczel's set theory, and the sections following provide a formal description and analysis of ST.

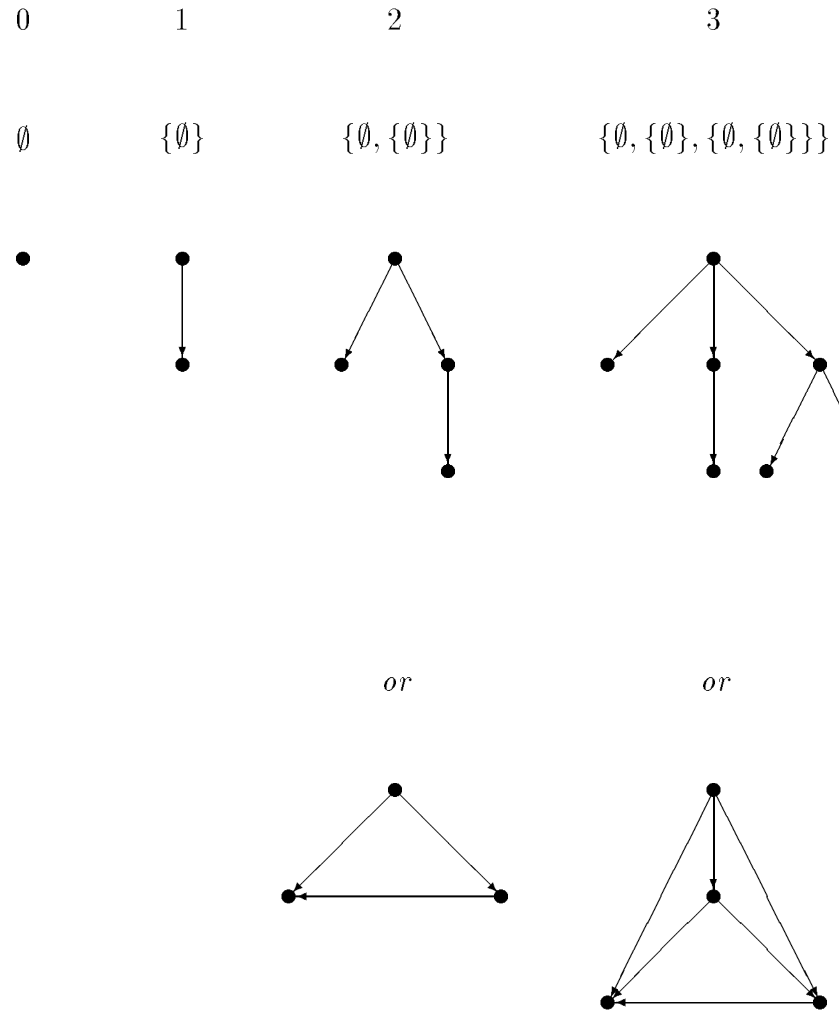


Figure 5: Apg representations of the natural numbers $0, 1, 2$ and 3.

3.1.1 Non-Well-Founded Set Theory

Aczel motivates his proposed *anti-foundation axiom*, which is to take the place of the regularity axiom, by developing a graph-based model of sets. Sets are identified with certain directed graphs called *accessible pointed graphs,* or apgs. Apgs are characterized by a distinguished node, the *point*, and by the property that every node in the graph is *accessible* from the point; there is a path from the point to every node in the graph. Intuitively, the point denotes the set being defined, while nodes represent members, and edges represent membership. Thus, the edge $b \longrightarrow a$ is interpreted as $a \in b$. Figure 5 presents the apgs corresponding to the ordinal representations of the natural numbers $0, 1, 2$ and 3. Note that 2 and 3 each have two apg representations given; in general, a set may give rise to many different apgs.

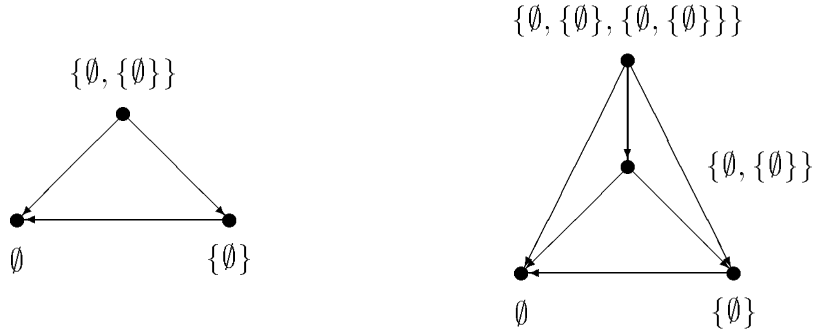


Figure 6: Pictures of 2 and 3.

A *decoration* of an apg 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 6 shows the 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. Aczel defines an apg to be *well-founded* exactly when it has no infinite path, and he notes the following.

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

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

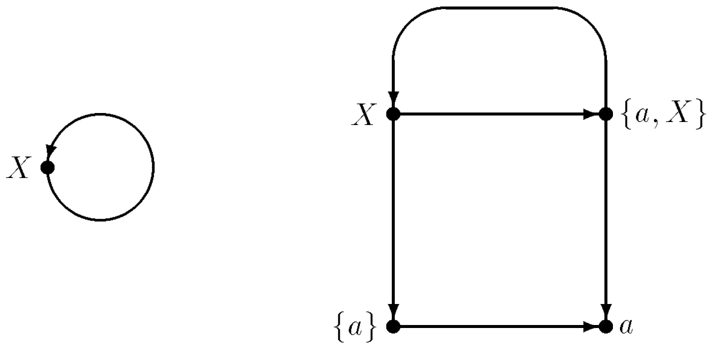
3. Every set has a picture.

He then states the anti-foundation axiom as follows.

Axiom 1 (Anti-Foundation) *Every apg has a unique decoration.*

Thus, every apg is the picture of a unique set, and as a consequence certain non-well-founded sets exist; in fact, any non-well-founded apg is the picture of a non-well-founded set. Figure 7 shows examples of non-well-founded sets and their pictures. The first picture corresponds to the set $\{\{\{...\}\}\}$, while the second corresponds to the "infinite stream" of a's, $\langle a, \langle a, ...\rangle\rangle$ (where a is left uninterpreted). 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, comprehension, collection, choice and anti-foundation is consistent, with apgs serving as the model. Let $ZFC^- + AFA$ denote this theory; for convenience, the axioms are listed in figure 8.



$X = \{X\}$ $\qquad$ $X = \langle a, X \rangle$

Figure 7: Non-well-founded sets and their pictures.

Extensionality	$\forall z.\, (z \in a \iff z \in b) \Rightarrow a = b$
Pairing	$\exists z.\, a \in z \wedge b \in z$
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	$\exists z.\, (\exists x \in z.\, \forall y.\, y \notin x) \wedge (\forall x \in z.\, \exists y \in z.\, x \in y)$
Comprehension	$\exists z.\, \forall x.\, x \in z \iff x \in a \wedge \phi$
Collection	$(\forall x \in a.\, \exists y.\, \phi) \Rightarrow \exists z.\, \forall x \in a.\, \exists y \in z.\, \phi$
Choice	$\forall x \in a.\, \exists y.\, y \in x \wedge \forall x_1, x_2 \in a.\, (\exists y.\, y \in x_1 \wedge y \in x_2 \Rightarrow x_1 = x_2)$
	$\Rightarrow \exists z.\, \forall x \in a.\, \exists! y \in x.\, y \in z$
Anti-foundation	*Every apg has a unique decoration.*

Figure 8: The axioms of $ZFC^- + AFA$.
a, b and c are arbitrary sets.
ϕ is any formula in which z does not appear free.

In addition to allowing the definition of new kinds of sets, 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. Let a *bisimulation* be a binary relation R on sets such that XRY implies that the following conditions are satisfied.

1. $\forall x \in X. \exists y \in Y. xRy$.

2. $\forall y \in Y. \exists x \in X. xRy$.

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 $X \equiv Y$ implies $X = Y$. Aczel shows the following (which we paraphrase).

Theorem 3.1 *The anti-foundation axiom is valid if and only if equality is strongly extensional.*

3.1.2 The Model

Using the non-well-founded set-theoretic machinery described above it is now possible to make sense of the informal description of the model of concurrency mentioned earlier. Using the comprehension and powerset axioms we define *finset*(X) for arbitrary X as follows. Let 2^X represent the set corresponding to X whose existence is guaranteed by the powerset axiom, and let *finite* denote the appropriately defined formula.

$$\mathit{finset}(X) = \{\, Z \in 2^X \mid \mathit{finite}(Z)\,\}$$

The anti-foundation axiom guarantees a unique solution to the following equation.

$$ST = \mathit{finset}(A \times ST)$$

This solution, which we call ST, defines our model of processes. The elements of ST are finite sets of pairs, the first element of which comes from A and the second element of which is an element of ST. Intuitively, an element of ST represents a process as a set of "possibilities," where a possibility consists of an action the process may perform and a continuation.

Using an alternative set theory to develop a model of processes may seem a rather drastic step. However, our reasons for doing so stem from the inability of standard set theory to allow us to define models for things that intuitively exist, and altering the axioms to suit our purposes, while maintaining the soundness of the underlying theory, seems justified. Moreover, the step is not so drastic as it seems, once one considers that the axioms of ZFC except for the regularity axiom are all still valid. In general, the fact that sets are not well-founded means that one cannot prove properties of all sets by induction on $\in$; however, properties of well-founded sets may still be established in this fashion. It is also worth noting that all sets definable in ZFC are definable in $ZFC^- + AFA$.

3.2 Developing CCS in the Model

This section gives an account of the CCS term constructors in ST by defining a function S from processes to ST. Assume that the set A satisfies the criteria presented previously for CCS action sets and that L is a relabeling as before. We also need the following operations on elements of ST. Let $a \in A$, $s, t \in ST$, and $f \in A \to B$ for some set B.

$$\begin{aligned}
prune(a,s) &= \{\, \langle b, prune(a,s') \rangle \mid \langle b, s' \rangle \in s \wedge b \neq a \,\} \\
map(f,s) &= \{\, \langle f(a), map(f,s') \rangle \mid \langle a, s' \rangle \in s \,\} \\
comp(s,t) &= \{\, \langle a, comp(s',t) \rangle \mid \langle a, s' \rangle \in s \,\} \cup \\
&\quad\ \{\, \langle b, comp(s,t') \rangle \mid \langle b, t' \rangle \in t \,\} \cup \\
&\quad\ \{\, \langle \tau, comp(s',t') \rangle \mid \exists a.\ \langle a, s' \rangle \in s \wedge \langle \overline{a}, t' \rangle \in t \,\}
\end{aligned}$$

These operators are well-defined operations on ST, since they involve applications of set constructions (namely, collection, comprehension and recursive definition) guaranteed by $ZFC^- + AFA$. In particular, in each case the uniqueness of the solution to the recursive equation for particular elements of ST is a result of AFA; for apgs s and t, action a and function f, it is possible to construct apgs corresponding to $prune(a,s)$, $map(f,s)$ and $comp(s,t)$.

Figure 9 inductively defines S for the nonrecursive process constructors. We now turn our attention to recursively defined processes. In order to define $S(\mathit{fix}(v.B))$ (where v is guarded in B) we shall apply AFA as follows. First, define $S(v)$ as v for variable v. Since v is guarded in B, it is possible to define an apg by identifying occurrences of v with the point of $S(B)$. AFA guarantees that the point of this apg corresponds to a unique set, and it is this set that we take as the definition of $S(\mathit{fix}(v.B))$. It is worth noting that this set may also be characterized as the unique solution to the equation $v = S(B)$.

In order to argue for the correctness of S it is necessary to define the CCS semantic relation $\xrightarrow{a}$ as a relation in terms of elements of ST. Given the intended interpretation of elements of ST, the process-based assertion $P \xrightarrow{a} P'$ should correspond to the ST-based assertion $\langle a, S(P') \rangle \in S(P)$. This is in fact the correct identification to make, as the following theorem shows.

Theorem 3.2 *The following are true.*

i. If $P \xrightarrow{a} P'$ *then* $\langle a, S(P') \rangle \in S(P)$.

$$
\begin{aligned}
S(nil) &= \emptyset \\
S(aP) &= \{\langle a, S(P)\rangle\} \\
S(P+Q) &= S(P) \cup S(Q) \\
S(P|Q) &= comp(S(P), S(Q)) \\
S(P\backslash a) &= prune(a, prune(\overline{a}, S(P))) \\
S(P[L]) &= map(L, S(P))
\end{aligned}
$$

Figure 9: The definition of S for nonrecursive process constructors.

ii. If $\langle a, s\rangle \in S(P)$ for some $s \in ST$, then there is a P' such that $P \xrightarrow{a} P'$ and $S(P') = s$.

Proof.

i. The proof proceeds by induction on the length of the proof needed to establish that $P \xrightarrow{a} P'$. The interesting case involves $P = \mathit{fix}(v.B)$. In this case, $B[P/v] \xrightarrow{a} P'$ by a shorter proof, whence $\langle a, S(P')\rangle \in S(B[P/v])$ by the induction hypothesis. Since P is the unique solution to the equation $v = S(B)$ it follows that $S(P) = S(B)[S(P)/v]$, and a simple inductive argument establishes that $S(B)[S(P)/v] = S(B[P/v])$.

ii. The proof proceeds by induction on the structure of P. Again, the interesting case is when $P = \mathit{fix}(v.B)$. Assume that $\langle a, s\rangle \in S(P)$. Since $S(P)$ is the unique solution to $v = S(B)$, $S(P) = S(B)[S(P)/v]$; therefore, there is a pair $\langle a, s'\rangle \in S(B)$ with $s'[S(P)/v] = s$. Using the induction hypothesis, there is a B' such that $B \xrightarrow{a} B'$ and $S(B') = s'$; therefore, $P \xrightarrow{a} B'[P/v]$. The result follows from the fact that $S(B'[P/v]) = S(B')[S(P)/v]$.

□

It is also possible to give an account of the SCCS parallel composition operator and some of the CSP operators [3]. To model the former, define a function *scomp* over ST as follows.

$$
\begin{aligned}
scomp(s, t) = \{\, &\langle a \circ b, scomp(s', t')\rangle \mid \\
&\langle a, s'\rangle \in s \wedge \langle b, t'\rangle \in t \,\}
\end{aligned}
$$

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

3.3 Equivalence in the Model

Equality in ST is strongly extensional, since ST is a set defined in a theory where the anti-foundation axiom is valid. The definition of bisimulation given by Aczel resembles the definition of strong bisimulation given for CCS processes, and this fact suggests that there is a relation between equality in ST and strong equivalence for CCS.

Theorem 3.3 *$P \sim Q$ if and only if $S(P) = S(Q)$.*

Proof. Suppose that $P \sim Q$. By the strong extensionality of equality in $ZFC^- + AFA$ it suffices to exhibit a bisimulation R such that $S(P)\ R\ S(Q)$. Define

$$
\begin{aligned}
R_1 &= \{ \langle S(P), S(Q) \rangle \mid P \sim Q \} \\
R_2 &= \{ \langle \{a\}, \{a\} \rangle \mid a \in A \} \\
R_3 &= \{ \langle a, a \rangle \mid a \in A \} \\
R_4 &= \{ \langle \{a, S(P)\} \{a, S(Q)\}, | \rangle a \in A \wedge P \sim Q \} \\
R &= R_1 \cup R_2 \cup R_3 \cup R_4
\end{aligned}
$$

As $\langle a, s \rangle$ is short-hand for $\{\{a\}, \{a, b\}\}$ (meaning that elements of ST contain elements of this form), it is easy to verify that R is a bisimulation such that $S(P)\ R\ S(Q)$.

Suppose that $S(P) = S(P')$, and let the relation $=_{ST}$ on processes be defined by $P =_{ST} Q$ when $S(P) = S(Q)$. To prove that $P \sim Q$ it suffices to show that $=_{ST}$ is a strong bisimulation. Suppose that $P \xrightarrow{a} P'$. By the previous theorem $\langle a, S(P') \rangle \in S(P)$, implying that $\langle a, S(P') \rangle \in S(Q)$, and by the previous theorem there is a Q' with $Q \xrightarrow{a} Q'$ and $S(P') = S(Q')$; that is, $P' =_{ST} Q'$. Therefore, $=_{ST}$ is a strong bisimulation. $\square$

4 Nuprl

Nuprl is a system that supports the interactive development of formal mathematics. On top of an expressive, albeit low-level, language for describing mathematical objects and reasoning about them, the system supplies an interactive *proof editor* that oversees the correct development of proofs. It also provides the user with flexible mechanisms for extending notation and building derived rules of inference to supplement the primitive notation and rules of inference from the theory on which the system is based. In the rest of this section we will briefly examine the Nuprl system and its underlying theory. The interested reader should consult [5] for a more complete account.

4.1 The Nuprl System

4.1.1 Editing Proofs

Nuprl supports the top-down development of proofs. The system represents proofs as trees of nodes, where a node contains a hypothesis list, a goal and a proof rule invocation and has as children nodes corresponding to subgoals generated automatically by the proof editor, *red*, on the basis of the form of the goal and the rule used. 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.

The following shows a single step in the proof in Nuprl of a lemma used to establish the fundamental theorem of arithmetic [13]. The lemma in question is a well-known theorem in number theory; it states that if a and b are relatively prime then there exist integers m and n such that

$$ma + nb = 1.$$

To prove this in Nuprl, one first invokes *red*; the system responds with the following window.

```
EDIT THM relative_primes
-----------------------------
?  top
<main proof goal>
```

To enter the main proof goal, we edit the slot `<main proof goal>` and enter the statement of the theorem, preceded by a `>>`. The system checks that the statement is well-formed and then prompts for a proof rule with the following.

```
EDIT THM relative_primes
-----------------------------
# top
>> ∀ k:{2..}.∀ a,b:N+
     where a+b < k & ∀ d:2...¬(d|a & d|b).
   ∃m,n:Int.  m*a + n*b = 1


BY <refinement rule>
```

Proof rules are called *refinement rules* because the system uses them to refine the goal into subgoals that need to be proven. In order to enter a rule, we edit the `<refinement rule>` and type the name of a rule. The system determines if the rule is applicable, and if it is it calculates the subgoals to be proven. In our example, we enter a rule setting up an inductive argument; the system's response appears in figure 10. There are two subgoals, indicated by `1#` and `2#`. Each subgoal consists of a numbered list of hypotheses, a `>>`, and a goal to be proven. In this case, the first subgoal has three hypotheses and a subgoal asking us to prove that the result holds for the base case (when `j=1 in int`), while the second subgoal has four hypotheses and a subgoal corresponding to the inductive case. The subgoals are proven by descending into them and using refinement rules in the same way. When a valid refinement rule results in no subgoals, the goal is complete.

```
EDIT THM relative_primes
# top
>> ∀ k:{2..}.∀ a,b:N+
     where a+b < j & ∀ d:2...¬(d|a & d|b).
   ∃m,n:Int.  m*a + n*b = 1

BY -- Intro THENW NonNegInductionUsing 1 1 ‘j‘

1# 1.  k:{2..}
   2.  j:int
   3.  j=1 in int
   >> ∀ k:{2..}.∀ a,b:N+
        where a+b < j-1 & ∀ d:2...¬(d|a & d|b).
      ∃m,n:Int.  m*a + n*b = 1

2# 1.  k:{2..}
   2.  j:int
   3.  1<j
   4.  ∀ k:{2..}.∀ a,b:N+
        where a+b < k & ∀ d:2...¬(d|a & d|b).
       ∃m,n:Int.  m*a + n*b = 1
   >> ∀ k:{2..}.∀ a,b:N+
         where a+b < j-1 & ∀ d:2...¬(d|a & d|b).
      ∃m,n:Int.  m*a + n*b = 1
```

Figure 10: An invocation of a proof rule.

4.1.2 Extending Notation

Nuprl provides a very flexible means for defining new notation. Using the *def* mechanism, it is possible to define new *display forms* in terms of existing ones. For example, in the example of the previous section, $\forall, \exists, \&$ and $|$ are all invocations of defs; they are not notations that are primitive to Nuprl.

Defs consist of two parts, a display form and a body. In the case of $\forall$, the def looks like this.

$$\forall\langle v : var\rangle : \langle t : type\rangle.\langle p : prop\rangle == \langle v\rangle : \langle t\rangle \rightarrow \langle p\rangle$$

The display form appears on the left of the $==$, the body on the right. Parameters are enclosed by $\langle\rangle$; these parameters must be instantiated when the definition is invoked. In the display form, parameters consist of a name (v, t or p in this example) and an optional comment describing the expected form of the argument. When the def is invoked, the display form appears on the screen, with $\langle\rangle$ indicating slots for the arguments. One enters arguments by typing directly on the indicated slots.

4.1.3 Defining Proof Rules

Nuprl also supplies a mechanism for building new proof rules from old ones. There is an interface between Nuprl and a variant of the ML programming language that has special types corresponding to Nuprl proofs and operators for manipulating objects of these types. The manipulation of proofs by these operators is guaranteed to preserve consistency because the operators can change proofs only by calling primitive proof rules in Nuprl.

In order to create derived proof rules (called *tactics*), one writes functions in ML that construct proofs. In the example appearing previously, `Intro THENW NonNegInductionUsing 1 1 ‘j‘` is a tactic that consists of two constituent tactics – `Intro` and `NonNegInductionUsing 1 1 ‘j‘` – connected by a *tactical*, or tactic-combining function, `THENW`. Both tactics were written by the developer of the proof, as was the tactical.

4.1.4 Other Features

Nuprl also supplies facilities for evaluating terms and storing theorems, defs, and the like. The Nuprl theory embodies a rudimentary programming language, and the evaluation mechanism allows the creation of environments in which terms in this language may be computed. The latter feature allows theorems, defs, tactics and evaluation environments to be saved from session to session. These collections are called *libraries* in Nuprl parlance.

4.2 The Nuprl Theory

We now turn our attention to describing the underlying theory. Like set theory, the Nuprl theory is a foundational theory of mathematics. Nuprl, however, embodies a *logical theory of types,* a theory based on enrichments of the notion of type encountered in many programming languages. These enrichments make the theory expressive enough to encode the constructive predicate calculus and versions of the classical predicate calculus via the "propositions-as-types" principle. Types of interesting objects can also be defined and the deductive machinery of the theory used to reason about properties of these objects. Other logical theories of types have been studied, the theory of Martin-Löf in [14] being a notable example.

The organization of our account is as follows. We first describe the *computation system* that underlies the Nuprl theory; terms in this system will be the terms to which types will be given. We then describe the types themselves and the rules that allow types and terms to be related. Following that, we explain how the predicate calculus may be interpreted in type theory and how proofs in the theory may also be interpreted as programs.

We should note that the Nuprl syntax that we use in the remainder of the paper differs from the syntax that appears in the system. We have done this to aid the readability of what follows.

4.3 Terms in Nuprl

Underlying the Nuprl type theory is a computation system, a collection of terms together with a set of rules describing their computational behavior. Owing to the syntactic complexity of Nuprl, the terms and rules for computing them are listed in the appendix; the remainder of this section is devoted to an examination of certain aspects of these terms.

Terms may be classified as either *canonical*, in which case no more computation may be done on them, or *noncanonical*, in which case computation is possible. The *evaluator* is the procedure that evaluates terms. If a term is canonical, then the evaluator terminates on this term. If the term is noncanonical, then the evaluator computes the value of certain subterms, called *principal arguments*, and using the result of these computations applies the computation rules to compute the value of the term. As an illustration (not taken from Nuprl) of these ideas, consider the way in which an expression like

$$\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2$$

is typically evaluated. First, B is computed, and based on the result of B either S_1 or S_2 is computed. In Nuprl jargon B would be the principal argument.

To illustrate the computation of an actual Nuprl term, consider the term

$$ind(5+4; u, v.0; 0; x, y.x + y).$$

As the table in the appendix indicates, the subterm $5 + 4$ is the principal argument, so the evaluator computes its value, 9. Since $9 > 0$ neither the second or the third subterms (separated by ";") are computed. Rather, the third subterm is; as the reduction rule in the appendix dictates, 9 is substituted for x, and the recursive invocation of the term with $9 - 1$ replacing 9 substituted for y, and the result of their addition computed. In this case, the final result is $9 + 8 + 7 + \ldots + 0 = 45$.

4.4 Types and Elements of Types

In order to reason about the computational behavior of the terms described above, the Nuprl theory defines a notion of *type*. Roughly speaking, a type includes a collection of canonical terms sharing a common structure and an equality relation on these terms; an arbitrary term has a particular type if it evaluates to a canonical term in the type. This section examines the type structure defined by Nuprl and presents the terms used to represent types.

Like sets, types are collections of elements. Built in to the notion of a type, however, is the notion of an equivalence relation that specifies when two elements in the type are equal. For example, a type of records might specify that two records are equal exactly when they have the same fields and equal values in corresponding fields.

In Nuprl, types are built from variables and base types using a collection of *type constructors*. Associated with each type constructor is a notion of equality that draws upon the equalities associated with the arguments to the type constructor and a collection of *value* constructors that describe the kinds of canonical terms that reside in, or *inhabit*, the type. For example, $\times$ is a type constructor that constructs the Cartesian product of its two arguments. Equality between elements in such a product type is pairwise equality, and the value constructor associated with $\times$ is $\langle , \rangle$, the pairing constructor.

Figure 11 lists the base types and the "data type" constructors (those that have analogs in many programming languages). These types are interpreted as follows. The type *void* contains no members. The type *int* is the type of integers. $A \times B$ is the Cartesian product of types A and B, with pairs of elements from these types being the canonical members of the newly constructed type. $A \mid B$ represents the "disjoint union" of types A and B; the elements of this type are the elements of A tagged with *inl* (signifying membership in the "left" type) and the elements of B tagged with *inr* (for the "right" type). Finally, $A \rightarrow B$ is the function space from A to B, with lambda terms as elements. We should note here that this account is not

Type	Members	Equality
void	–	–
int	$0, 1, -1, \ldots$	integer equality
$A \times B$	$\langle a, b\rangle$ for $a \in A, b \in B$	$\langle a, b\rangle = \langle a', b'\rangle \ in\ A \times B$ $\iff a = a' \ in\ A \wedge b = b' \ in\ B$
$A \mid B$	$inl(a), inr(b)$ for $a \in A, b \in B$	$inl(a) = inl(a') \ in\ A \mid B \iff a = a' \ in\ A$ $inr(b) = inr(b') \ in\ A \mid B \iff b = b' \ in\ B$
$A \rightarrow B$	$\lambda x.\ b$ where $b[a/x] \in B$	$\lambda x.\ b = \lambda y.\ b' \ in\ A \rightarrow B$ $\iff \forall a \in A.\ b[a/x] = b[a/y] \ in\ B$

Figure 11: Base types and data types in Nuprl.

rigorous; in particular, we rely on the reader's intuition about functions and substitution in the description of $A \rightarrow B$.

Nuprl also includes two generalized versions of data type constructors not usually found in typed programming languages, as well as a scheme for recursive type construction. These constructors appear in figure 12 and are listed together because each type has in it a *type expression*, B, containing a free variable x, such that when x is replaced by any value from an appropriate index type, a type results. The type $\sum_{x \in A} B$ represents the indexed sum of B with respect to A; its elements are pairs consisting of an element, or "tag", from the index set A and an element from the corresponding type defined by the type expression B. Similarly, the type $\prod_{x \in A} B$ represents the indexed product of B with respect to A. Elements of this type are "sequences", indexed by A, of elements from the appropriate type defined by B. These sequences are represented as functions from the index type, A, the type expression B, with the type of the result of the function application depending on the value of the argument. Finally, $rec(x.B)$ is a recursive type. The elements of this type depend on the type constructors that appear in B, as does the equality between elements. For example, the type $rec(x.int \mid (x \times x))$, which corresponds to a type of ordered binary trees with integer-valued leaves, has elements consisting of $inl(i)$ for $i \in int$ and $inr(\langle t_1, t_2\rangle)$ for trees t_1 and t_2, and equality is defined in the obvious way.

The final group of types we consider here can be characterized as "logical" types in that understanding them depends to a certain extent on seeing how types correspond to propositions. This correspondence is explored in the next section; for the time being we assume the existence of "predicates" that are either true or false of elements in a type. Let P be such a predicate; figure 13 lists these types. Equality types like $a = a' \ in\ A$ are empty if a and a' are not equal in A; otherwise, they have the element *axiom* as their only member. The

Type	Members	Equality
$\sum_{x\in A} B$	$\langle a,b\rangle$ for $a \in A, b \in B[a/x]$	$\langle a,b\rangle = \langle a',b'\rangle\ in\ \sum_{x\in A} B$ $\iff a = a'\ in\ A \wedge b = b'\ in\ B[a/x]$
$\prod_{x\in A} B$	$\lambda x.\ b$ where $b[a/x] \in B[a/x]$	$\lambda x.\ b = \lambda y.\ b'\ in\ \prod_{x\in A} B$ $\iff \forall a \in A.\ b[a/x] = b'[a/y]\ in\ B[a/x]$
$rec(x.B)$		

Figure 12: Dependent type and recursive type constructors.

Type	Members	Equality
$a = a'\ in\ A$	$axiom$	
$\{x : A \mid P\}$	$a \in A$ where $P(a)$	$a = a'\ in\ \{x : A \mid P\} \iff a = a'\ in\ P$
$(x,y) : A//P$	$a \in A$	$a = a'\ in\ (x,y) : A//P \iff P(a,a')$

Figure 13: Logical type constructors.

"set type" $\{x : A \mid P\}$ consists of elements of A for which P is true, while the "quotient" type $(x,y) : A//P$ consists of elements of A, with equality redefined so that two members a and b are equal exactly when $P[a/x, b/y]$ is true. P must be an equivalence relation for this definition to make sense.

Two other kinds of types are referred to in what follows. The first, "infinite types," allow the definition of infinite objects such as streams or infinite depth trees. The theory of these types has been worked out [15], but they are not yet implemented in the system. The second, "universes", allow collections of types to be treated as types. The only universe that appears in what follows is U_1, which is the type of all types (excluding itself) that have been presented so far.

4.5 Propositions as Types

In order to understand how mathematics can be formalized in Nuprl one must first understand how logic is interpreted in the theory. The representation of inference rules is treated in the next section; here we concentrate on giving a type-theoretic semantics to propositions in the predicate calculus. The propositions-as-types principle [6, 12] gives such a semantics for the constructive predicate calculus by identifying the notion of truth, by which propositions are given meaning, with the notion of *type inhabitation*. Just as one asks whether a proposition is true, one may ask whether a type contains a member. This modeling of truth forms the core of the interpretation of propositions as types, for once this identification is made the standard propositional connectives (under their constructive intepretation) correspond directly with

Propositions	Types
false	*void*
$P \wedge Q$	$P \times Q$
$P \vee Q$	$P \mid Q$
$P \Rightarrow Q$	$P \rightarrow Q$
$\neg P$	$P \rightarrow void$
$\forall x : A.\ B$	$\prod_{x \in A} B$
$\exists x : A.\ B$	$\sum_{x \in A} B$

Figure 14: Propositions as types.

type constructors just presented.

Figure 14 describes the interpretation of propositions as types. The proposition *false* is never true, and the type *void* never has any members, so *false* is interpreted as *void*. $P \wedge Q$ is true when both P and Q are true, and $P \times Q$ contains a member exactly when both P and Q do, so one may sensibly interpret $\wedge$ as $\times$. Constructively, $P \vee Q$ holds when one (or both) of P and Q hold and one can demonstrate which; $P \mid Q$ is inhabited when either P or Q, or both, has a member, and the form of the member (*inl* or *inr*) indicates which type contributes the member, so $\vee$ may be modeled as $|$. The proposition $P \Rightarrow Q$ is true when, under the assumption of the truth of P, Q holds, and $P \rightarrow Q$ has an element precisely when there exists a function that converts an element of P into an element of Q. One constructive account of $\neg P$ is $P \Rightarrow$ *false*, implying the type-theoretic intepretation given in the figure. The proposition $\forall x : A.\ B$ holds when for each $a \in A, B[a/x]$ holds, just as $\prod_{x \in A} B$ contains a member when for each $a \in A, B[a/x]$ contains a member. Finally, $\exists x : A.\ B$ holds constructively when an $a \in A$ may be exhibited such that $B[a/x]$ is true, just as $\sum_{x \in A} B$ contains an element when there is a pair $\langle a, b\rangle$ with $a \in A$ and $b \in B[a/x]$.

It is also possible to provide Nuprl interpretations of classical proposition constructors. Classically, $P \vee Q$ holds if P or Q holds; this is equivalent to $\neg(\neg P \wedge \neg Q)$, and this statement can be interpreted in Nuprl using the scheme described above. Similarly, the statement $\exists x : A.\ B$ is equivalent to $\neg \forall x : A.\ \neg B$.

Constructive mathematics is controversial, owing to the well-known fact that the law of the excluded middle is not an axiom. This does not mean that constructivists claim this axiom is always false, but rather that in order to use it one must first prove it as a theorem. In the preceding account of constructive logic, the proposition $P \vee \neg P$ holds precisely when a decision procedure for P exists. In the development of CCS that follows, operations and predicates of interest are all decidable, so the absence of the law of the excluded middle does not prove burdensome. There are instances, however, where the semantics of logic supplied by Nuprl require that theorem statements be altered somewhat. In particular, on several occasions a proposition P is not be provable, but the proposition $\downarrow P$, where

$$\downarrow P = \{\, 0 = 0 \ \mathit{in}\ \mathit{int} \mid P \,\},$$

is. The reasons for this have to do with the definition of the formal semantics of the Nuprl theory and are somewhat technical; the interested reader is referred to [5] for a full discussion. Intuitively, $\downarrow P$ is the type containing exactly one element (*axiom*) when P is "true" (i.e. contains an element) and no elements otherwise. P and Q can be different as types, but if they are both either inhabited or not inhabited then $\downarrow P$ and $\downarrow Q$ are equal types. It is this property of squashed types that sometime enables $\downarrow P$ to be provable even though P is not.

4.6 Deduction in Nuprl

Given that propositions may be interpreted as types, the next stage in representing logic in Nuprl involves describing how theorems may be proven. Associated with each type constructor is a collection of inference rules that describe the correct constructions for the type. For example, if $a \in A$ and $b \in B$ then we should be able to conclude that $\langle a, b\rangle \in A \times B$; a variant of this rule is in fact one of the rules associated with $\times$.

Figure 15 lists some of the type inference rules of Nuprl. Rules are sequent based; the expressions appearing to the left of $\vdash$ are assumptions, and those appearing to the right are conclusions. One readily apparent feature of the Nuprl logic is that it is a *refinement logic*, meaning that inference rules are applied to goals, with subgoals being specified as necessary in order to establish the truth of the goal. Although it is beyond the scope of this paper to list all the rules of Nuprl, the rules correspond in a natural way with the inference rules found in the predicate calculus.

Another important feature of proofs in Nuprl is that they correspond to *programs* in the following sense. The constructive interpretation of propositions given above identifies propositions with types, and given the identification made between truth (of a proposition) and inhabitation (of a type), a proof of a proposition interpreted as a type specifies the construction of an element in the type. This is particularly interesting when the proposition being proven has the form $P \Rightarrow Q$ or $\forall x : A.\ B$, since elements of these types are functions. There is also a facility built into the Nuprl theory for gaining access to the computational content of the proof of a theorem. The term *term_of*($name$) has as its value the term corresponding to the proof of the theorem $name$.

$$\Sigma\text{-intro}\qquad \frac{H \vdash \sum_{x\in A} B}{\begin{array}{l} H \vdash a\ \mathit{in}\ A \\ H \vdash B[a/x] \\ H, y : A \vdash B[y/x]\ \mathit{in}\ U_i \end{array}}$$

$$\Sigma\text{-elim}\qquad \frac{H, z : \sum_{x\in A} B, H' \vdash C}{H, z : \sum_{x\in A} B, H', u : A, v : B[u/x], z = \langle u, v\rangle\ \mathit{in}\ \sum_{x\in A} B \vdash C[\langle u, v\rangle/z]}$$

$$\Pi\text{-intro}\qquad \frac{H \vdash \prod_{x\in A} B}{\begin{array}{l} H, y : A \vdash B[y/x] \\ H \vdash A\ \mathit{in}\ U_i \end{array}}$$

$$\Pi\text{-elim}\qquad \frac{H, f : \prod_{x\in A} B, H' \vdash C}{\begin{array}{l} H, f : \prod_{x\in A} B, H' \vdash a\ \mathit{in}\ A \\ H, f : \prod_{x\in A} B, H', y : B[a/x], y = f(a)\ \mathit{in}\ B[a/x] \vdash C \end{array}}$$

Goals appear above subgoals; there is one subgoal per line.

H, H' are lists of assumptions.

a, f, u, v, y, z and U_i are parameters supplied by the user.

Figure 15: Sample Nuprl rules.

5 Implementing CCS in Nuprl

This section describes a Nuprl library containing a development of a finitary version of ST and of nonrecursive CCS terms in this version of ST. The necessity of the finiteness restriction stems from the fact that our method for implementing ST requires the infinite type constructors mentioned previously, and as these are not yet implemented we are obliged to use the recursive type constructor instead. We shall have more to say about this point later.

The library itself contains 240 Nuprl objects. The first 50 or so comprise definitions of standard logic connectives and quantifiers in terms of the underlying type theory. The next 150 objects develop the finite set theory necessary to give an account to give an account of the ST model, and the last 40 objects present a development of CCS.

The implementation makes frequent use of a convention for defining new type-theoretic objects that we briefly describe here. The Nuprl type theory is not decidable, meaning that in general it is impossible to determine algorithmically whether a given term has a given type. Furthermore, there is no means in the Nuprl def 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 T) in proofs of theorems involving defined objects, we frequently introduce such an object, such as *finset*, by proving a theorem whose name is the object followed by an underscore (in our example, *finset_*), whose statement is the type of the object being defined and whose first proof step involves explicitly introducing the term corresponding to the object. The statement of *finset_*, for example, is $U_1 \rightarrow U_1$, since the *finset* constructor maps a type to the type of its finite sets, 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 create the definition *finset*, then, one creates a def whose body is *term_of(finset_)*. The system understands this convention, and whenever it sees the notation *finset* it knows 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 proven automatically.

The remainder of the section is organized in the following fashion. Sections 5.1, 5.2 and 5.3 describe the implementation of *finset*, ST and CCS, respectively. It should be noted here that all Nuprl objects quoted in the figures throughout this section are faithful transcriptions from the actual Nuprl library.

5.1 Implementing finite sets

As we saw in the previous chapter, the model ST rests on a set-theoretic foundation. Since Nuprl supplies type theory as a foundational language, 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 concepts like union and then prove that the associated axioms in set theory are true of our implementation.

5.1.1 Defining *finset*

Representing a type constructor corresponding to sets of elements of a given type in a natural and efficient fashion, as well as a way that enables its use in recursive type expressions, 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. However, restrictions on the bodies of recursive types prohibit this implementation's use in recursive type expressions. On the other hand, identifying sets with lists of elements introduces inefficiencies in the implementation of set operations.

To circumvent these difficulties we 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 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 needed to define a type of finite sets for an arbitrary type in U_1; in what follows types in U_1 will be referred to as "small types". Table I lists some basic Nuprl definitions, and table II lists associated theorem statements. These are described in the following paragraphs. 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 the type of the corresponding initial segment of the positive integers. That is,

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

The Nuprl definition *Interval* defines the notation $\{1 \ldots n\}$ as $\mathit{term\_of}(\mathit{Interval\_})(n)$. For an

Name	Display form	Body
p1	$\langle p\rangle.1$	$spread(\langle p\rangle; u, v.u)$
p2	$\langle p\rangle.2$	$spread(\langle p\rangle; u, v.v)$
Interval	$\{1\ldots\langle n\rangle\}$	$term\_of(Interval\_)(\langle n\rangle)$
set_eq	$\langle s\rangle =_e \langle t\rangle\ over\ \langle T\rangle$	$term\_of(\langle T\rangle)(\langle s\rangle)(\langle t\rangle)$
finset	$finset(\langle T\rangle)$	$term\_of(finset\_)(\langle T\rangle)$

Table I: Nuprl definitions.

Name	Statement
*Interval_*	$int \rightarrow U_1$
*set_eq_*	$\prod_{T\in U_1} T\rightarrow T\rightarrow U_1$
*finset_*	$U_1 \rightarrow U_1$

Table II: Nuprl theorems

arbitrary small type T, the collection of arrays of type T may be represented as the type $\sum_{n\in int}\{1\ldots n\}\rightarrow T$, which defines the type of pairs mentioned previously.

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

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

The definition *set_eq_* then defines the notation $s =_e t\ over\ T$ to be $term\_of(T)(s)(t)$.

The next three theorems in the library prove that $s =_e t\ over\ T$ is an equivalence relation relation for any small type T. The objects *set_eq_reflexive*, *set_eq_commutative* and *set_eq_transitive* establish that $s =_e t\ over\ T$ is reflexive, commutative and transitive, respectively; these theorems are used in the definition of *finset*.

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. As *set_eq* is an equivalence relation, this is well-defined. Thus,

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

5.1.2 Membership and extensionality

In an array representation of finite sets the membership predicate for an element a and a set s may state that there is an index i such that $s[i] = a$. Our implementation adopts this approach. The most obvious candidate for the membership predicate,

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

cannot be proven to be a function in $\prod_{T \in U_1} T \rightarrow \mathit{finset}(T) \rightarrow U_1$, since different representatives of the same set yield types which are not equal; however, the term

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

can, and the Nuprl theorem *in_* does so. The definition *in* defines the notation $a \in_T s$ as *term_of*(*in_*)$(T)(a)(s)$.

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

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

The more usual version, of course, is not squashed and contains the "only if" portion of the bi-implication. We elected to state the theorem in this form because there is some work involved in proving the "if" part of the bi-implication, owing to the fact that one may not always substitute equals for equals in Nuprl. Moreover, 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.

5.1.3 Null and singleton sets

The axioms of $ZFC^- + AFA$ in figure 8 specify the existence of a null set (in the infinity axiom) and the existence of pairs. The Nuprl implementation also implements the null set, but instead of implementing pairs it implements singleton sets. We choose to do this for reasons of efficiency; singleton sets appear frequently in the description of subsequent set-theoretic constructions and in the implementation of CCS that follows, and having singleton sets implemented efficiently is highly desirable.

In an array-based description of sets empty arrays may correspond to the null set, while arrays consisting of one element may correspond to singleton sets. Our implementation follows

Name	Display Form	Body
null	$\emptyset_{\langle T\rangle}$	$term\_of(null\_)(\langle T\rangle)$
singleton	$\{\langle a\rangle\}_{\langle T\rangle}$	$term\_of(singleton\_)(\langle T\rangle)(\langle a\rangle)$

Table III: Nuprl definitions for null and singleton sets.

Name	Statement
*null_*	$\prod_{T\in U_1} finset(T)$
*singleton_*	$\prod_{T\in U_1} T \rightarrow finset(T)$

Table IV: Nuprl typing theorems for null and singleton sets.

this convention; for an arbitrary small type T, the Nuprl array $\langle 0, \lambda x.\ x\rangle$ denotes the null set, and for an arbitrary a in T $\langle 1, \lambda x.\ a\rangle$ denotes the singleton set containing a. Table III lists the definitions associated with the null set and singleton sets, and table IV lists the associated typing theorems.

The proofs of the appropriate axiomatic descriptions of these sets are straightforward; the theorem statements are listed in figure 16. The theorem *null_axiom* states that for any small type T $\emptyset_T$ has no elements. Theorem *singleton_axiom_1* states that if two elements in an arbitrary small type T are equal then 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.

null_axiom

$\forall T : U_1.\ \forall a : T.\ \neg(a \in_T \emptyset_T)$

singleton_axiom_1

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

singleton_axiom_2

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

Figure 16: Axiomatizations of the null set and singleton sets.

Name	Display Form	Body
union	$\langle s\rangle \cup_{\langle T\rangle} \langle t\rangle$	*term_of(union_)*$(\langle T\rangle)(\langle s\rangle)(\langle t\rangle)$
set_union	$\cup_{\langle T\rangle}\langle s\rangle$	*term_of(set_union_)*$(\langle T\rangle)(\langle s\rangle)$
comprehension	$\{\, x \in \langle s\rangle \mid \langle P\rangle \wedge \langle p\rangle \,\}_T$	*term_of(comprehension_)*$(\langle T\rangle)(\langle P\rangle)(\langle p\rangle)(\langle s\rangle)$
collection	$map\ \langle f\rangle(\langle T\rangle : \langle U\rangle)\ on\ \langle s\rangle$	*term_of(collection_)*$(\langle T\rangle)(\langle U\rangle)(\langle f\rangle)(\langle s\rangle)$

Table V: Nuprl definitions for union, family union, comprehension and collection.

Name	Statement
*union_*	$\prod_{T\in U_1} finset(T) \to finset(T) \to finset(T)$
*set_union_*	$\prod_{T\in U_1} finset(finset(T)) \to finset(T)$
*comprehension_*	$\prod_{T\in U_1} \prod_{P\in(T\to U_1)}$ $\prod_{p\in(P\ is\ decidable\ on\ T} finset(T) \to finset(T)$
*collection_*	$\prod_{T\in U_1} \prod_{U\in U_1} (T\to U) \to finset(T) \to finset(U)$

Table VI: Nuprl typing theorems for union, family union, comprehension and collection.

5.1.4 Union

The next three sections define and axiomatize the set-theoretic constructions of union, comprehension and collection. In each case the appropriate array constructions are presented, and the axioms corresponding to the constructions are proven correct. The Nuprl definitions and typing theorems corresponding to these operations are grouped together and appear in tables V and VI.

Two notions of set union appear in the implementation. The first is a binary union operator that takes two finite sets as arguments and returns their union. The second takes one argument, a finite set of finite sets, and returns the union of the sets in the argument. This latter version is closer to the notion of union as presented in the $ZFC^- + AFA$ axiomatization; however, in the absence of a pairing set constructor both are necessary.

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 array concatenation in our framework involves some subtlety because array bounds can be negative; one must do more than just add together the bounds in order to compute the bound for the concatenation of the two arrays. The implementation circumvents this obstacle with the *union_axiom_1*

$\forall T : U_1. \forall s, t : \mathit{finset}(T). \forall a : T.\ a \in_T s \cup_T t \Rightarrow\ \downarrow (a \in_T s \vee a \in_T t)$

union_axiom_2

$\forall T : U_1. \forall s, t : \mathit{finset}(T). \forall a : T.\ (a \in_T s \vee a \in_T t) \Rightarrow a \in_T s \cup_T t$

Figure 17: The axiomatization of union.

definition of a function on the integers, *nat*, written $\uparrow i$ for integers i. If $i \leq 0$ then $\uparrow i = 0$; otherwise, $\uparrow i = i$. The following term now computes the concatenation of two arrays.

$$\lambda s, t.\ \langle \uparrow s.1 + \uparrow t.1, \lambda i.\ \mathit{less}(i; s.1; s.2(i); t.2(i - s.1)) \rangle$$

The array bound of result is the sum of the "naturalized" array bounds of the input arrays, and the enumeration function of the result first compares its argument (an index) to determine which of its arguments' enumeration functions to use. The Nuprl theorem *union_* establishes that this term has the correct type.

The theorems *union_axiom_1* and *union_axiom_2* are listed in figure 17; they prove that the implementation satisfies the properties of binary union. The first of these states that if a is in $s \cup_T t$ then it is in either s or in t, while the second states the converse.

The "family union" operator is defined recursively in terms of the binary union operator. In the case of an empty set of sets, the null set is returned; otherwise, the binary union of the "first" set in the set of sets and the family union of the rest of the set of sets is returned. The following term performs precisely this computation. It should be noted that since s is an array representation of a set of sets, its enumeration function, $s.2$, maps integers to finite sets.

$$\lambda s.\ \mathit{ind}(s.1; u, v.\emptyset_T; \emptyset_T; u, v.s.2(u) \cup_T v)$$

If the bound, $s.1$, is negative then $\emptyset_T$ is returned. If the bound is 0 then $\emptyset_T$ is returned. If the bound is positive then the union of the u^{th} set in s, $s.2(u)$, and the result of the recursive invocation, v, is returned. Theorem *set_union_* establishes that this term is correctly typed, while the theorems listed in figure 18 establish the correctness of the implementation.

5.1.5 Comprehension

Given a predicate P on the elements of an array, one can derive a set-theoretic comprehension scheme in the following fashion. For a set 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, do not.

$\underline{\textit{set\_union\_axiom\_1}}$
$\forall T : U_1.\ \forall s : \mathit{finset}(\mathit{finset}(T)).\ \forall a : T.\ (\exists t : \mathit{finset}(T).\ t \in_{\mathit{finset}(T)} s \wedge a \in_T t) \Rightarrow a \in_T \cup_T s$

$\underline{\textit{set\_union\_axiom\_2}}$
$\forall T : U_1.\ \forall s : \mathit{finset}(\mathit{finset}(T)).\ \forall a : T.\ a \in_T \cup_T s \Rightarrow \downarrow (\exists t : \mathit{finset}(T).\ t \in_{\mathit{finset}(T)} s \wedge a \in_T t)$

Figure 18: The axiomatization of family union.

The Nuprl development of a comprehension operator follows this algorithm, but it differs in one major respect. In Nuprl functions in the standard function space types must be total. However, for general predicates (where, in Nuprl, a predicate over elements of type T is a function from T to 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 Nuprl definition *decidable* defines the notation P *is decidable on* T for arbitrary predicates P and types T to be exactly this; hence, the type of the Nuprl term corresponding to set comprehension is as indicated in table V. The third argument to a term of this type must be a proof that P is decidable; this proof is used as a decision procedure in the construction of the desired set.

The following term implements the algorithm described above for set comprehension.

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

Given a predicate P, a decision procedure p for P, and an array s, the body of the term inductively analyzes the array bound, $s.1$, of s. If $s.1 \leq 0$ then $\emptyset_T$ is returned, since s has no elements. If $s.1 > 0$ then the last element in the set, $s.2(u)$, is tested using p. If the result is an *inl*( )term then P must hold (since the type of p is $P \vee \neg P$) and the set returned is $\{s.2(u)\}_T \cup v$, where v represents the result of recursive invocation. Otherwise, v is returned.

Theorems *comprehension_axiom_1* and *comprehension_axiom_2* establish that the comprehension scheme enjoys the desired properties. These are listed in figure 19.

5.1.6 Collection

Our implementation of sets gives rise to a particularly simple implementation of a collection scheme. Given a function f of type $T \rightarrow U$ for small types T and U and an array s, the mapping of f onto s can be achieved by composing f and $s.2$ for the enumeration function of

$\underline{\textit{comprehension\_axiom\_1}}$
$\forall T : U_1.\ \forall P : (T \rightarrow U_1).\ \forall p : P\ \textit{is decidable on}\ T.\ \forall s : \textit{finset}(T).\ \forall a : T.$
$a \in_T s \wedge P(a) \Rightarrow a \in_T \{\, x \in s \mid P \wedge p \,\}_T$

$\underline{\textit{comprehension\_axiom\_2}}$
$\forall T : U_1.\ \forall P : (T \rightarrow U_1).\ \forall p : P\ \textit{is decidable on}\ T.\ \forall s : \textit{finset}(T).\ \forall a : T.$
$a \in_T \{\, x \in s \mid P \wedge p \,\}_T \Rightarrow a \in_T s \wedge P(a)$

Figure 19: The axiomatization of comprehension.

$\underline{\textit{collection\_axiom\_1}}$
$\forall T, U : U_1.\ \forall f : T \rightarrow U.\ \forall s : \textit{finset}(T).\ \forall a : T.\ a \in_T s \Rightarrow f(a) \in_U \textit{map}\ f(T : U)\ \textit{on}\ s$

$\underline{\textit{collection\_axiom\_2}}$
$\forall T, U : U_1.\ \forall f : T \rightarrow U.\ \forall s : \textit{finset}(U).\ \forall b : U.$
$b \in_U \textit{map}\ f(T : U)\ \textit{on}\ s \Rightarrow\ \downarrow \exists a : T.\ f(a) = b\ \textit{in}\ U \wedge a \in_T s$

Figure 20: The axiomatization of collection.

the result. The term

$$\lambda f.\ \lambda s.\ \langle s.1, \lambda x.\ f(s.2(x)) \rangle$$

realizes this scheme. Tables V and VI give the definition and type corresponding to this term, while the theorems in figure 20 prove its correctness.

5.2 Implementing *ST*

As we mentioned earlier, the implementation of *ST* is in fact an implementation of the finite, or well-founded, elements of *ST* as presented in section 3. This implementation uses the recursive type constructor in conjunction with *finset* to define a type corresponding to finitary *ST*.

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 types of actions to types corresponding to *ST* for the type of actions. However, the implementation described here does not follow this course, primarily because the algebraic formalisms are interesting only when, for example, equality between actions is decidable. Therefore, we elect to model actions as

$\underline{\mathit{finset\_to\_ST}}$
$\forall s : \mathit{finset}(\mathit{Action} \times ST).\ s\ \mathit{in}\ ST$

$\underline{\mathit{ST\_to\_finset}}$
$\forall s : ST.\ s\ \mathit{in}\ \mathit{finset}(\mathit{Action} \times ST)$

Figure 21: Unrolling theorems for ST.

integers, and accordingly, the definition of *action* defines the notation *Action* to correspond to the Nuprl type *int*.

The typing theorem *st_* establishes that the following term is a type.

$$\mathit{rec}(st.\ \mathit{finset}(\mathit{Action} \times st))$$

This type consists of finite sets of pairs of actions and elements of the type being defined. The Nuprl definition *st* defines the notation ST to be *term_of*(*st_*). The two theorems listed in figure 21 allow elements in ST to be unrolled and rerolled.

5.3 Implementing CCS

We now describe the implementation of CCS in the Nuprl version of ST. The type *int* corresponds to CCS action set, with 0 representing τ and $\overline{a}$ represented as $-a$. On occasion we refer to members of ST as sets of *branches*, where a branch has type $\mathit{Action} \times ST$. We also say that action a *heads* branch $\langle a, s\rangle$ and that s is the *continuation* of this branch.

We should note at the outset that we do not implement recursive processes (because of the finitary nature of the implementation of ST) or relabeling. However, our experience with implementing the other operators indicates that developing the relabeling construct is straightforward. With these caveats in mind, tables VII and VIII list the CCS operators that are implemented and the associated typing theorems.

In the description of implementations that follow, theorems are mentioned that establish the semantic correctness of the operators defined. In particular, the relation $\xrightarrow{a}$ appears frequently in these theorems. The definition *arrow* defines the notation $\langle s\rangle \xrightarrow{\langle a\rangle} \langle t\rangle$ as *term_of*(*arrow_*)($\langle s\rangle$)($\langle a\rangle$)($\langle t\rangle$), where theorem *arrow_* establishes that the term

$$\lambda s.\ \lambda a.\ \lambda t.\ \langle a, t\rangle \in_{\mathit{Action} \times ST} s$$

has type $ST \rightarrow \mathit{Action} \rightarrow ST \rightarrow U_1$.

Name	Display Form	Body
nil	*NIL*	*term_of*(*nil_*)
prepend	$\langle a\rangle\langle P\rangle$	*term_of*(*prepend_*)($\langle a\rangle$)($\langle P\rangle$)
plus	$\langle P\rangle + \langle Q\rangle$	*term_of*(*plus_*)($\langle P\rangle$)($\langle Q\rangle$)
merge	$\sum\langle s\rangle$	*term_of*(*merge_*)($\langle s\rangle$)
prune	$\langle P\rangle - \langle a\rangle$	*term_of*(*prune_*)($\langle a\rangle$)($\langle P\rangle$)
restriction	$\langle P\rangle\backslash\langle a\rangle$	*term_of*(*restriction_*)($\langle a\rangle$)($\langle P\rangle$)
composition	$\langle P\rangle\vert\langle Q\rangle$	*term_of*(*composition_*)($\langle P\rangle$)($\langle Q\rangle$)

Table VII: Nuprl definitions of CCS process constructors.

Name	Statement
*nil_*	ST
*prepend_*	$Action \rightarrow ST \rightarrow ST$
*plus_*	$ST \rightarrow ST \rightarrow ST$
*merge_*	$finset(ST) \rightarrow ST$
*prune_*	$Action \rightarrow ST \rightarrow ST$
*restriction_*	$Action \rightarrow ST \rightarrow ST$
*composition_*	$ST \rightarrow ST \rightarrow ST$

Table VIII: Nuprl typing theorems for CCS process constructors.

5.3.1 The process constructors *nil*, prefixing, + and $\sum$

The four operators defined here have very straightforward implementations in ST. The first of the operators, NIL, corresponds to $\emptyset_{Action \times ST}$. The theorem *nil_* establishes that this term inhabits ST, and definition *nil* defines the appropriate notation. The theorem *nil_semantics* listed in figure 22 proves the correctness of this representation. The proof follows immediately from *null_axiom*.

Prefixing an action a to an existing process P is represented by the singleton set $\{\langle a, P\rangle\}_{Action \times ST}$. Theorem *prepend_* establishes the type correctness of the term

$$\lambda a.\ \lambda P.\ \{\langle a, P\rangle\}_{Action \times ST},$$

and the definition *prepend* defines the appropriate notation. Theorem *prepend_semantics* is listed in figure 22 and proves the validity of this construction; the proof follows from the theorem *singleton_axiom_1*.

Summing two processes P and Q corresponds to taking the union of their ST representations. Theorem *plus_* verifies that this operation is type correct, and definition *plus* defines the appropriate notation. Theorems *plus_semantics_1* and *plus_semantics_2* establish that this implementation is semantically correct. The proofs of these theorems follow from *union_axiom_2*.

Finally, the sum of a finite set of processes corresponds to the set union operator. Theorem *merge_* proves that the term

$$\lambda s.\ \cup_{Action \times ST}\, s$$

has type $\mathit{finset}(ST) \rightarrow ST$, and definition *merge* defines the appropriate notation. Using *set_union_axiom_1*, theorem *merge_semantics* in figure 22 establishes that *merge* is correct.

5.3.2 Restriction

The representation of restriction is more complicated than the representations of the previously described CCS operators because this operator has a recursive semantic specification; thus, the Nuprl term corresponding to restriction must be defined recursively. If a process P is represented as $\sum_i a_i P_i$, then $P \backslash a$ may be represented equationally in the following fashion.

$$P \backslash a = \sum_{\{i \mid a_i \neq a \vee a_i \neq \overline{a}\}} a_i (P_i \backslash a)$$

This recursive description may be translated into a Nuprl term, as the remainder of this section shows.

$\underline{\textit{nil\_semantics}}$
$\forall P : ST.\ \forall a : Action.\ \neg(NIL \xrightarrow{a} P)$

$\underline{\textit{prepend\_semantics}}$
$\forall P : ST.\ \forall a : Action.\ aP \xrightarrow{a} P$

$\underline{\textit{plus\_semantics\_1}}$
$\forall P, Q, P_1 : ST.\ \forall a : Action.\ P \xrightarrow{a} P_1 \Rightarrow P + Q \xrightarrow{a} P_1$

$\underline{\textit{plus\_semantics\_2}}$
$\forall P, Q, Q_q : ST.\ \forall a : Action.\ Q \xrightarrow{a} Q_1 \Rightarrow P + Q \xrightarrow{a} Q_1$

$\underline{\textit{merge\_semantics}}$
$\forall s : \textit{finset}(ST).\ \forall P_1 : ST.\ \forall a : Action.\ (\exists P : ST.\ P \in_{ST} s \wedge P \xrightarrow{a} P_1) \Rightarrow \sum s \xrightarrow{a} P_1$

Figure 22: Semantic theorems for NIL, prefixing, $+$ and $\sum$.

The implementation of restriction is given in terms of an operator, *prune*, that recursively eliminates branches headed by the indicated action. Equationally, *prune* satisfies the following.

$$prune(a, P) = \sum_{\{\, i | a_i \neq a \,\}} a_i(prune(a, P_i))$$

This definition indicates that using comprehension is appropriate, so the first step in implementing *prune* involves defining and proving decidable the pruning predicate, $x.1 \neq a$ for branch x. Theorem *prune_predicate_* proves that the term

$$\lambda a.\ \lambda x.\ \neg x.1 = a \textit{ in } Action$$

has type $Action \rightarrow (Action \times ST) \rightarrow ST$, meaning that for any action a, this term applied to a yields a predicate. The definition *prune_predicate* defines the notation

$$\neg \textit{ action part } \neq \langle a \rangle,$$

and theorem *prune_predicate_decidable* establishes that the pruning predicate is decidable:

$$\forall a : Action.\ (\textit{action part } = a) \textit{ is decidable over } Action \times ST.$$

The term in figure 23 computes *prune*. On action a and ST P this term applies the recursive invocation of the function being defined (*prune* represents the recursive invocation in the term)

$$\begin{array}{l} \lambda a.\ \lambda s.\ rec\_ind(s;\ prune, t. \\ map\ \lambda b.\ \langle b.1, prune(b.2)\rangle\ (Action \times ST : Action \times ST) \\ on\ \{\, x \in t \mid \neg action\ part\ = a \wedge \\ term\_of(prune\_predicate\_decidable)(a)\,\}_{Action \times ST} \end{array}$$

Figure 23: The Nuprl term for *prune*.

prune_semantics

$\forall P, P_1 : ST.\ \forall a, b : Action.\ a \neq b \Rightarrow P \xrightarrow{a} P_1 \Rightarrow (P - b) \xrightarrow{a} (P_1 - b)$

restriction_semantics

$\forall P, P_1 : ST.\ \forall a, b : Action.\ a \neq b \Rightarrow a \neq \overline{b} \Rightarrow P \xrightarrow{a} P_1 \Rightarrow P \backslash b \xrightarrow{a} P_1 \backslash b$

Figure 24: Semantic theorems for *prune* and restriction.

to the continuation of each branch in P, with branches headed by a removed. Theorem *prune_* proves that this term has type $Action \rightarrow ST \rightarrow ST$, and definition *prune* defines the following notation.

$$\langle P \rangle - \langle a \rangle$$

in terms of the extraction term from *prune_*. Theorem *prune_semantics* establishes the semantics of this operator; it is listed in figure 24.

The implementation of restriction is straightforward in terms of *prune*; $P \backslash a$ is just $(P - a) - \overline{a}$. Theorem *restriction_* establishes the appropriate type of the term

$$\lambda a.\ \lambda s.\ (s - a) - \overline{a},$$

and definition *restrict* defines the appropriate notation. Theorem *restriction_semantics* is listed in figure 24.

5.3.3 Composition

The CCS composition operator has a straightforward equational description given by the expansion theorem. If $P = \sum_i a_i P_i$ and $Q = \sum_j b_j Q_j$ then

$$P|Q = \sum_i a_i(P_i|Q) + \sum_j b_j(P|Q_j) + \sum_{a_i = b_j} \tau(P_i|Q_j).$$

However, giving a Nuprl account of composition involves some complexity that belies this simple equational specification. The complexity results from a certain inflexibility in the

Nuprl recursion scheme. The most natural definition of $P|Q$ involves a simultaneous induction on P and Q; however, *rec_ind* forms only allow the inductive analysis of one object at a time. On the other hand, we can rewrite the equational specification of composition in terms of two mutually recursive function, *rcomp* and *lcomp*, that are each recursive in only one of the two arguments to the composition operator. Consider the following.

$$\begin{aligned} lcomp(P) &= \lambda Q.\ rcomp(P, Q) \\ rcomp(P, Q) &= \sum_i a_i lcomp(P_i)(Q) + \\ &\quad \sum_j b_j rcomp(P, Q_j) + \\ &\quad \sum_{a_i = \overline{b_j}} \tau lcomp(P_i)(Q_j) \end{aligned}$$

Let $P|Q$ be defined as $lcomp(P)(Q)$. By substituting $\lambda Q.\ rcomp(\_, Q)$ for $lcomp(\_)$ in the second equation the validity of this specification of can be verified. It is also the case that *lcomp* in inductively defined only in terms of P, while *rcomp* is recursively defined only in terms of Q. This specification therefore suggests a form for the Nuprl term, with two nested recursive forms implementing the mutually recursive definitions of *lcomp* and *rcomp* and the body of the inner recursive specification using the collection and comprehension operators defined previously to implement *rcomp*.

The definition of composition to follow uses comprehension twice, with two different predicates. The first, *prune_predicate*, has already been proven decidable; the second is the negation of *prune_predicate*. Theorem *weed_* proves that the term corresponding to this predicate,

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

has type $Action \rightarrow (Action \times ST) \rightarrow U_1$, and the definition *weed* defines the notation

$$action\ part\ = \langle a \rangle$$

as $term\_of(weed\_)(\langle a \rangle)$.

The term contained in figure 25 corresponds to the composition operator. This term behaves in the following fashion. Roughly speaking, the outer *rec_ind* form defines *lcomp*, while the inner form defines *rcomp*. In each case, the term being recursively analyzed in each form appears first, followed by a semicolon; the two variables following semicolon in each case represent the recursion variable and a variable to which the term preceding the semicolon is bound. The body of the inner *rec_ind* form consists of three terms separated by $+$'s; these three terms correspond to the three parts of the recursive definition of *rcomp* in the equation specification given previously. The first of these terms computes $\sum_i a_i lcomp(P_i, Q)$

$$
\begin{array}{l}
\lambda s.\ rec\_ind\ (s; lcomp, s_1. \\
\quad \lambda t.\ rec\_ind\ (t; rcomp, t_1. \\
\quad\quad map\ (\lambda b.\ \langle b.1, lcomp(b.2)(t_1)\rangle)\ on\ s_1 + \\
\quad\quad map\ (\lambda b.\ \langle b.1, rcomp(b.2)\rangle)\ on\ t_1 + \\
\quad\quad \sum\ map\ (\lambda c.\ map\ (\lambda b.\ \langle 0, lcomp(b.2)(c.2)\rangle)\ on \\
\quad\quad\quad\quad\quad\quad \{\, x \in \{\, x \in s_1 \mid \neg action\ part = 0 \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad term\_of\,(prune\_predicate\_decidable)(0)\,\} \\
\quad\quad\quad\quad\quad\quad\quad \mid\ action\ part = -c.1 \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad term\_of\,(weed\_decidable)(-c.1)\,\}) \\
\quad\quad\quad on\ t_1))
\end{array}
$$

Figure 25: The Nuprl term for composition.

in terms of the collection operator. The function $\lambda b.\ \langle b.1, lcomp(b.2)(t_1)\rangle$ is "mapped" over s_1; the resulting element of ST consists of branches headed by actions heading branches in s_1 and having continuations that are the composition of t_1 and continuations of branches in s_1. Similarly, the second of these terms calculates $\sum_j b_j rcomp(P, Q_j)$. In this case the mapping function is $\lambda b.\ rcomp(b.2)$; it should be noted that s_1 does not appear in the application of *rcomp* to $b.2$ since $rcomp(\_)$ in this term corresponds to $rcomp(P, \_)$ in the equational description.

The last of the three terms computes $\sum_{a_i=\overline{b_j}} \tau\, lcomp(P_i, Q_j)$. As this term is complex, it is helpful to analyze it piece by piece. The $\sum$ constructor indicates that the remainder of the term is to return a set S of elements of ST. S is computed using an application of the collection operator to t_1, implying that the mapping function should map a branch to an element of ST. Given a branch c, this function builds an element of ST essentially by first pruning away all branches in $s.1$ that are headed either by τ (0, in our representation), or by actions other than the inverse of the action heading c, then returning a set branches headed by τ and having as continuations the composition of the continuation of c and the continuations of the "pruned" s_1. Thus, once this outermost application of collection is complete, a set of elements of ST remains, where each element of ST consists of branches headed by τ. The $\sum$ operator then collapses these into one.

Figure 26 lists the theorems proven to establish the semantic correctness of this implementation. The first theorem deserves special notice, as it is a statement of the expansion theorem. This theorem is used to prove the other three.

$\underline{\mathit{expansion\_thm}}$

$$
\begin{aligned}
&\forall P, Q : ST.\\
&P|Q = map\ (\lambda b.\ \langle b.1, P|b.2\rangle)\ on\ Q\ +\\
&\qquad map\ (\lambda b.\ \langle b.1, b.2|Q\rangle)\ on\ P\ +\\
&\qquad \textstyle\sum\ map\ (\lambda c.\ map(\lambda b.\ b.2|c.2)\ on\\
&\qquad\qquad\qquad \{\, x \in \{\, x \in P \mid \neg action\ part = 0\ \wedge\\
&\qquad\qquad\qquad\qquad\qquad term\_of(prune\_predicate\_decidable)(0)\,\}\\
&\qquad\qquad\qquad\quad \mid\ action\ part\ = -c.1 \wedge term\_of(weed\_decidable)(-c.1)\,\})\\
&\qquad\quad on\ Q\ \ in\ ST
\end{aligned}
$$

$\underline{\mathit{composition\_semantics\_1}}$

$$\forall P, Q, Q_1 : ST.\ \forall a : Action.\ Q \xrightarrow{a} Q_1 \Rightarrow P|Q \xrightarrow{a} P|Q_1$$

$\underline{\mathit{composition\_semantics\_2}}$

$$\forall P, Q, P_1 : ST.\ \forall a : Action.\ P \xrightarrow{a} P_1 \Rightarrow P|Q \xrightarrow{a} P|Q_1$$

$\underline{\mathit{composition\_semantics\_3}}$

$$\forall P, Q, P_1, Q_1.\ \forall a : \{\, x : Action \mid x \neq 0\,\}.\ P \xrightarrow{a} P_1 \wedge Q \xrightarrow{-a} Q_1 \Rightarrow P|Q \xrightarrow{0} P_1|Q_1$$

Figure 26: Semantic theorems for composition.

equality_semantics_1

$\forall P, Q : ST.\ (\forall a : Action.$
$(\forall P_1 : ST.\ P \xrightarrow{a} P_1 \Rightarrow \exists Q_1.\ Q \xrightarrow{a} Q_1 \wedge P_1 = Q_1\ in\ ST) \wedge$
$(\forall Q_1 : ST.\ Q \xrightarrow{a} Q_1 \Rightarrow \exists P_1 : ST.\ P \xrightarrow{a} P_1 \wedge P_1 = Q_1\ in\ ST))$
$\Rightarrow P = Q\ in\ ST$

equality_semantics_2

$\forall P, Q : ST.$
$P = Q\ in\ ST \Rightarrow$
$(\forall a : Action.$
$(\forall P_1 : ST.\ P \xrightarrow{a} P_1 \Rightarrow \exists Q_1.\ Q \xrightarrow{a} Q_1 \wedge P_1 = Q_1\ in\ ST) \wedge$
$(\forall Q_1 : ST.\ Q \xrightarrow{a} Q_1 \Rightarrow \exists P_1 : ST.\ P \xrightarrow{a} P_1 \wedge P_1 = Q_1\ in\ ST))$

Figure 27: Semantic theorems for equality.

5.4 Equality

The theorems *equality_semantics_1* and *equality_semantic_2* characterize equality in ST with respect to the definition of bisimulation. These theorems are listed in figure 27.

6 Using Nuprl to Reason about CCS

We now examine using the framework developed in the previous section in conjunction with the rest of Nuprl to analyze the properties of processes expressed in CCS. Two approaches are considered, and examples of each are presented. In the first, Nuprl terms that compute the traces corresponding to elements of ST are defined and used to show that an implementation of a binary semaphore is correct. In the second, a development of a compositional proof system for the Hennessy-Milner logic due to Stirling [18] is sketched. We have not implemented this axiomatization, since the range of properties expressible in HML is limited; however, many of the issues involved in implementing a modal logic in Nuprl – defining a semantics for propositions, writing tactics that implement the inference rules – are issues that we handle.

$\underline{traces}$

$\lambda s.\ rec\_ind(s; lister, t.int\_eq(t.1; 0;$

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

$\quad \cup_{int\ list}\ (map\lambda b.\ map\lambda l.\ (b.1).l\ on lister(b.2)\ on\ t)))$

$\underline{list\_set}$

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

$\underline{list\_traces}$

$\lambda s.\ list\_set(traces(s))$

Figure 28: Nuprl functions for computing traces.

6.1 Analyzing Traces – The Mutual Exclusion Problem

By using the Nuprl facilities for term evaluation it is possible to write functions that allow one to analyze the traces of elements of ST. The rest of this section describes these functions, which exist in the Nuprl evaluator object *test*, and an application of these functions to an implementation of a binary semaphore defined by Milner [16].

For our purposes, a *trace* of an element s in ST is a maximal path through s. The functions that compute the traces of s are listed in figure 28. The first of these functions, *traces*, takes as input an element s of ST and returns a finite set comprising the traces of s. This function is defined recursively; the term converts s into a set of sets of traces, one set of traces for each branch of s, and then coalesces this set of sets of traces into a set of traces using $\cup_{int\ list}$. (Recall that the type *Action* and the type *int* are identical and that "." is the Nuprl list constructor.) The second of these functions takes a finite set s and returns a list of the elements of s; this function provides a convenient way of viewing a set. The final function is the composition of the first two functions and generates a list of the traces of s.

Now consider the following finitary version of the mutual exclusion problem. Let P_1 and P_2 be processes with critical sections ab and cd, respectively, for $a, b, c, d \in Action$. In order to ensure that ab and cd do not interfere with each other we define a process sem that corresponds to a semaphore. Consider the following CCS definitions.

$$\begin{aligned} P_1 &= pabvnil \\ P_2 &= pcdvnil \\ sem &= \overline{pvpv}nil \end{aligned}$$

Using the expansion theorem, it is easy to verify that the system $(P_1|sem|P_2)\backslash P\backslash V$ enforces the mutual exclusion of ab and cd.

These processes are represented as stated in the Nuprl library object *mutex_defs*. The library object *mutex_eval* then evaluates the term $list\_traces(P_1|sem|P_2)\backslash P\backslash V$ and binds the result to identifier l. The value of l,

$$(0.a.b.0.0.c.d.0).(0.c.d.0.0.a.b.0).nil,$$

establishes that mutual exclusion is enforced.

6.2 Implementing Hennessy-Milner Logic

The Hennessy-Milner logic (HML) was developed in order to give a modal characterization of observational equivalence [10]. As a logic for expressing properties of programs it is not very interesting; however, it has served as a testbed for methods of developing complete axiomatizations for modal logics for reasoning about CCS and SCCS, and so we consider a sketch of an implementation of one of these axiomatizations here. The axiomatization is due to Stirling [18]; he provides a complete axiomatization for HML with respect to a fragment of SCCS that does not have have a restriction operator. It should be noted that Stirling has also recently developed a complete axiomatization for full SCCS [19]; however, we do not consider that here. As before, we also do not consider nonterminating processes.

As with the implementation of CCS described above, our approach to modeling HML is semantic-based. Using ST as the domain of individuals and the encoding of the predicate calculus in Nuprl, we give a type-theoretic account of the semantics of HML propositions. Then we define the satisfaction relations given by Stirling and indicate how tactics can be written that simulate Stirling's inference rules. If carried out in Nuprl, the result of this effort would be an incremental proof checker for Stirling's axiomatization, with tactics being entered as inference rules and the Nuprl proof editor automatically generating the appropriate subgoals.

6.2.1 Syntax and semantics of HML

The syntax of HML propositions is presented in figure 29, and the semantics is defined in figure 30. The semantics is specified in terms of elements P of the set of processes. The modal operators are indexed by elements of $Action$.

In order to give an account of HML in Nuprl we define the semantics of HML propositions in terms of type inhabitation. When the predicate calculus is modeled in type theory propositions are identified with types, and the satisfaction relation (or the "notion of truth") on propositions is identified with type inhabitation; a proposition is true when the corresponding type has an element in it. We adopt the same approach here, with one important distinction. In the

$$
\begin{array}{rcl}
\phi & ::= & T \mid F \\
 & & \phi \wedge \phi \mid \phi \vee \phi \\
 & & \langle a \rangle \phi \mid [a]\phi
\end{array}
$$

Figure 29: The syntax of Hennessy-Milner logic. $a \in \mathit{Action}$

$$
\begin{array}{ll}
P \models T & \text{for all processes } P. \\
P \models F & \text{for no process } P. \\
P \models \phi_1 \wedge \phi_2 & \text{if } P \models \phi_1 \text{ and } P \models \phi_2. \\
P \models \phi_1 \vee \phi_2 & \text{if } P \models \phi_1 \text{ or } P \models \phi_2. \\
P \models \langle a \rangle \phi & \text{if } \exists P' \in \{\, P' \mid P \xrightarrow{a} P' \,\}.\ P' \models \phi. \\
P \models [a]\phi & \text{if } \forall P' \in \{\, P' \mid P \xrightarrow{a} P' \,\}.\ P' \models \phi.
\end{array}
$$

Figure 30: The semantics of Hennessy-Milner logic.

predicate calculus, the satisfaction relation is unary; $\models$ is defined over formulas only. In the case of modal logics in general, however, and HML in particular, $\models$ is binary; it is defined over processes and formulas. Accordingly, instead of modeling HML formulas as types we model them as type-valued functions over ST, and the clause $P \models \phi$ is interpreted as $N(\phi)(P)$, where $N(\phi)$ is the Nuprl interpretation of ϕ. Figure 31 details the semantic development of HML in Nuprl.

6.2.2 Defining provability

Stirling defines two notions of provability: $\vdash$ and $\vdash_\phi$ for HML formula ϕ. The former corresponds to the usual notion. The latter introduces a notion of context into provability: $P \vdash_\phi \psi$ exactly when for every Q such that $Q \vdash \phi$, $P \otimes Q \vdash \psi$.

Providing Nuprl translations for these notions is straightforward. The notion $P \vdash \phi$ becomes $\vdash_{Nuprl} N(\phi)(P)$; in general we write $\vdash$ instead of $\vdash_{Nuprl}$ when the meaning is clear from context. The notion $P \vdash_\phi \psi$ becomes $\vdash \forall Q : \{Q : ST \mid N(\phi)(Q)\}.\ N(\psi)(P \otimes Q)$.

6.2.3 Implementing inference rules

We turn now to the implementation of Stirling's axiomatization. As his system consists of seven axioms and 23 inference rules we do not present a complete development here; a full account may be found in [3]. Instead, we examine selected inference rules and discuss how they may be implemented as tactics.

HML formula	Nuprl translation
T	$\lambda P.\ 0 = 0\ in\ int$
F	$\lambda P.\ void$
$\phi_1 \wedge \phi_2$	$\lambda P.\ (N(\phi_1)(P) \wedge N(\phi_2)(P))$
$\phi_1 \vee \phi_2$	$\lambda P.\ N(\phi_1)(P) \vee N(\phi_2)(P)$
$\langle a \rangle \phi$	$\lambda P.\ \downarrow \exists y : Action\#ST.\ y \in P \wedge y.1 = a\ in\ Action \wedge N(\phi)(y.2)$
$[a]\phi$	$\lambda P.\ \downarrow \forall y : Action\#ST.\ y \in P \wedge y.1 = a\ in\ Action \Rightarrow N(\phi)(y.2)$

Figure 31: Nuprl semantics for HML formulas.
$N(\phi)$ represents the translation of formula ϕ.
P is an element of ST.

Axiom	Translation
$P \quad \vdash \quad T$	$\vdash \quad 0 = 0\ in\ int$
$nil \quad \vdash \quad [a]\phi$	$\vdash \quad \forall B \in nil.\ B.1 = a\ in\ Action \Rightarrow \phi(B.2)$
$P \quad \vdash_\phi \quad T$	$\vdash \quad \forall Q : \{t : ST \mid \phi(t)\}.\ 0 = 0\ in\ int$
$P \quad \vdash_F \quad \phi$	$\vdash \quad \forall Q : \{t : ST \mid void\}.\ \phi(P \otimes Q)$
$nil \quad \vdash_\phi \quad [a]\psi$	$\vdash \quad \forall Q : \{t : ST \mid \phi(t)\}.\ \forall B \in nil \otimes Q.$ $B.1 = a\ in\ Action \Rightarrow \psi(B.2)$
$aP \quad \vdash \quad [b]\phi$ if $b \neq a$	$\vdash \quad \forall a, : bAction.\ b \neq a \Rightarrow$ $\forall B \in aP.\ B.1 = c\ in\ Action \Rightarrow \phi(B.2)$
$aP \quad \vdash_\phi \quad [b]\psi$ if $b \backslash a$ does not exist	$\vdash \quad \forall a : Action.\ \forall Q : \{\,t \mid ST\}\phi(t).\ \forall B \in aP \otimes Q.$ $B.1 = b\ in\ Action \Rightarrow \psi(B.2)$

Table IX: Stirling's axioms for HML.

Rule	Translation
$P+Q \vdash \langle a\rangle\phi$ $P \vdash \langle a\rangle\phi$	$\vdash \exists B \in P+Q.\ B.1 = a\ in\ Action \wedge \phi(B.2)$ $\vdash \exists B \in P.\ B.1 = a\ in\ Action \wedge \phi(B.2)$
$P+Q \vdash \langle a\rangle\phi$ $Q \vdash \langle a\rangle\phi$	$\vdash \exists B \in P+Q.\ B.1 = a\ in\ Action \wedge \phi(B.2)$ $\vdash \exists B \in Q.\ B.1 = a\ in\ Action \wedge \phi(B.2)$
$P+Q \vdash_\phi \langle a\rangle\psi$ $P \vdash_\phi \langle a\rangle\psi$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \exists B \in (P+Q)\otimes R.$ $B.1 = a\ in\ Action \wedge \psi(B.2)$ $\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \exists B \in P\otimes R.$ $B.1 = a\ in\ Action \wedge \psi(B.2)$
$P+Q \vdash_\phi \langle a\rangle\psi$ $Q \vdash_\phi \langle a\rangle\psi$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \exists B \in (P+Q)\otimes R.$ $B.1 = a\ in\ Action \wedge \psi(B.2)$ $\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \exists B \in Q\otimes R.$ $B.1 = a\ in\ Action \wedge \psi(B.2)$

Table X: Stirling's $+\langle\rangle I$ rules.

The axioms of Stirling's system appear in table IX. These axioms are all provable as theorems in Nuprl.

We now examine two representative groups of Stirling's inference rules. In each case we present the rules in the group as refinement rules, with the conclusion on top and the subgoals underneath. We then give the Nuprl "translation" of the goal and each of the subgoals of the rule. Then we discuss how tactics may be written that, given proofs of the subgoals, construct proofs of the goal.

Table X lists the $+\langle\rangle I$ rules. These rules allow conclusions involving propositions of the form $\langle a\rangle\phi$ to be made about SCCS terms of the form $P+Q$ on the basis of the behavior of P and Q.

The first two rules follow in a simple fashion from the definition of $+$ as $\cup$; by using theorem *union_axiom_2* as a lemma it is straightforward to establish that the subgoal implies the body. The third and fourth rules rely on the definition on $\otimes$. Assuming the proof of the expansion theorem for $\otimes$, the reasoning to be implemented in this tactic involves the following. An arbitrary R of type ST satisfying ϕ is assumed (using an application of the $\forall$-introduction rule), and the subgoal allows us to conclude that there is a B in $P\otimes R$ satisfying the appropriate property. Using the expansion theorem for $\otimes$, we may conclude, with the help of the appropriate axiom for $\cup$, that this B is also in $(P+Q)\otimes R$. Thus, the subgoal implies the goal. A symmetrical line of reasoning justifies the fourth rule.

Table XI lists Stirling's $\otimes I$ rules. These rules allow properties to be deduced about SCCS

Rule	Translation
$P \otimes Q \vdash \ \psi$	$\vdash \psi(P \otimes Q)$
$P \vdash \ \phi$	$\vdash \phi(P)$
$Q \vdash_{\phi} \psi$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \psi(Q \otimes R)$
$Q \otimes P \vdash \ \psi$	$\vdash \psi(P \otimes Q)$
$P \vdash \ \phi$	$\vdash \phi(P)$
$Q \vdash_{\phi} \psi$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \psi(Q \otimes R)$
$P \otimes Q \vdash_{\phi} \gamma$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \gamma(P \otimes Q \otimes R)$
$P \vdash_{\phi} \psi$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \psi(P \otimes R$
$Q \vdash_{\psi} \gamma$	$\vdash \forall R : \{t : ST \mid \psi(t)\}.\ \gamma(Q \otimes R)$
$Q \otimes P \vdash_{\phi} \gamma$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \gamma(P \otimes Q \otimes R)$
$P \vdash_{\phi} \psi$	$\vdash \forall R : \{t : ST \mid \phi(t)\}.\ \psi(P \otimes R$
$Q \vdash_{\psi} \gamma$	$\vdash \forall R : \{t : ST \mid \psi(t)\}.\ \gamma(Q \otimes R)$

Table XI: Stirling's $\otimes I$ rules.

terms of the form $P \otimes Q$ on the basis of the properties of P and Q.

These rules have straightforward implementations. The first and second rules involve applying the second subgoal to P; the first subgoal establishes that in fact this application is justified. The third and fourth rules involve first assuming an arbitrary R of type ST satisfying ϕ. The first subgoal allows us to conclude that $P \otimes R$ satisfies ψ. Applying the second subgoal to $P \otimes R$ then allows us to establish that $P \otimes R \otimes Q$ satisfies γ, and using the expansion theorem it can be proven that $P \otimes R \otimes Q = P \otimes Q \otimes R \ in \ ST$.

6.2.4 A Hypothetic HML Session in Nuprl

In order to demonstrate the desirability of implementing a logic for reasoning about CCS or SCCS in Nuprl a short example of the style of interaction such an implementation would have is now presented. We assume the existence of tactics for each of Stirling's rules and appropriate defs for the formula constructors provability relations.

The example we consider is a finitary version of one in [18]. The goal is to establish that

$$a(bnil + cnil) \otimes \overline{a} 1 nil \otimes 1 \overline{b} nil$$

satisfies

$$[1]\langle 1 \rangle T.$$

First, one types in the goal, and the system responds for a request for an inference rule.

```
EDIT THM ex1
>> a(bnil + cnil) x (-a)1nil x 1(-b)nil |- [1]<1>T

BY <refinement rule>
```

One types in the rule `stxI#1` with parameter `'[a]<1>T'`, which would be a tactic corresponding to the first proof rule in table XI. The system responds with the subgoals as indicated.

```
EDIT THM ex1
>> a(bnil + cnil) x (-a)1nil x 1(-b)nil |- [1]<1>T

BY xI#1 '[a]<1>T'

1# >> a(bnil + cnil) |- [a]<b>T

2# >> (-a)1nil x 1(-b)nil |-([a]<b>T) [1]<1>T
```

One could then descend into the subgoals, using the proof rules to successively refine the goals into instances of the axioms. It would also be possible to write a tactic that automatically recognized instances of the axioms and proved them; this tactic could then automatically be invoked by other tactics on the subgoals they generate.

An important thing to note is the way in which the Nuprl system enforces the correctness of the proof steps. In the "interaction" just described, the only things the user would have typed in were the rules; the system generates the subgoals automatically. Another thing to note is that with judicious use of Nuprl definitions, the syntax of the objects being handled can be made very natural. Thus, even though the underlying Nuprl syntax bears no relation to SCCS the proof just described resembles very closely the proof that Stirling gives.

Of course, given a logic developed as HML is above, there is nothing to prevent users of Nuprl from writing tactics that use the implementations of the primitive inference rules of the logic to construct more complex proof rules corresponding to larger, more natural deductions. The feasibility of doing this has been demonstrated in other Nuprl applications.

7 Conclusions and Directions for Research

This paper examines one approach to how one might use a system like Nuprl to reason about concurrency in general, and CCS in particular. Our approach involves developing a model of concurrency in the Nuprl language and using the tools available in Nuprl to reason about the behavior of objects in the model. We have tried to indicate how a formal theory of mathematics, together with an implementation of the theory that incorporates tools for developing proofs and encapsulating notation, can be used to reason about concurrency in a rigorous, machine-checked fashion. We consider one example of a process whose behavior is checked using tools developed in Nuprl for calculating traces, and we develop a more speculative case for encoding logics for reasoning about CCS as collections of tactics that manipulate the objects of the underlying semantic theory. No logic is as yet implemented, owing to the lack of expressive logics for reasoning about CCS, but we indicate how Stirling's axiomatization of the Hennessy-Milner logic may be implemented.

Our experience with Nuprl suggests that it is partially successful for the task we undertook. The facilities for encapsulating notation and defining proof rules proved to be invaluable; without them, the level of detail at which we would have been required to work at would have made the project impossible. The *def* facility, however, has some irksome features; in particular, substituting into a *def* causes the notation to be lost. Unless a great deal of care is taken, this can lead to sudden increases in the size of expressions one is manipulating. The single biggest drawback to using Nuprl for our purposes, however, involves the efficiency of the implementation. The Nuprl terms corresponding to elements in ST, even for the simple semaphore example, became extremely large (owing to Nuprl's lazy evaluation mechanism); it took 10 minutes of computation on a LISP machine to compute the traces for this example. This clearly prohibits the analysis of more interesting examples. However, these problems do not appear insurmountable, and it may be the case that by using later versions of Nuprl our work could provide the basis for a true interactive verification environment for CCS.

Several directions for future work are apparent. Developing an expressive logic to implement as we have indicated is one interesting goal to pursue. Another involves a style of reasoning that we have not addressed in this paper, and that is the equational style usually associated with CCS. Since equality in the ST model corresponds to strong equivalence in CCS, the axioms that are true of strong equivalence are true of equality in ST and hence provable as theorems in Nuprl. Developing such an axiomatic theory would be of interest. Another interesting direction that could be pursued involves decision procedures. In particular, equality in ST is decidable, and proving this fact in Nuprl would yield a decision procedure for equality. Using this decision procedure, then, it would be possible to test automatically whether a process is equivalent to its specification.

Perhaps the whole point to this exercise is that using Nuprl it is possible to build an environment for concurrency in which a variety of different kinds of reasoning about processes are possible. The advantage to using Nuprl is that correctness is always guaranteed.

Acknowledgements

The authors would like to thank Doug Howe for his invaluable assistance in the development of the Nuprl library described in section 5.

A Terms in Nuprl

Terms in Nuprl fall into two categories: *canonical* and *noncanonical*. Canonical terms denote values; when the evaluator terminates, it returns a canonical term. Noncanonical terms have as values the canonical terms returned by the evaluator. Table XII lists the canonical terms in Nuprl, and table XIII lists the noncanonical terms. In each case a short description of the intended meaning of the term is also presented.

One evaluates noncanonical terms by first evaluating their principal arguments and then applying the appropriate reduction rule, if one exists. If one does not exist, an error results. Table XIV lists the reduction rules for noncanonical terms. An appropriate definition of substitution is assumed.

Term	Meaning
i	
x	
nil	Empty list.
$t_1.t_2$	Lists with head t_1 and tail t_2.
$inl(t), inr(t)$	Left and right members of disjoint union types.
$\langle t_1, t_2 \rangle$	Pairs.
$\lambda x.\ t$	Functions.
$axiom$	Member of equality and less types.
$void$	A type with no elements.
int	The type of integers.
$t\ list$	The type of lists with elements from t.
$t_1 \mid t_2$	The disjoint union of t_1 and t_2.
$\sum_{x \in t_1} t_2$	Dependent product.
$\prod_{x \in t_1} t_2$	Dependent function.
$\{x : t_1 \mid t_2\}$	The type of elements of t_1 satisfying t_2.
$(x, y) : t_1 // t_2$	The type of elements of t_1 with equality t_2.
$rec(x.t)$	Recursively defined type.
$t_1 = t_2\ in\ t_3$	Equality types.
$i < j$	"Less than" types.
U_i	Universes.

Table XII: Canonical terms in Nuprl.
(Subscripted) t's denote terms.
x, y denote variables.
i, j denote integer constants.

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.
$\underline{t_1}/\underline{t_2}$	Division.
$\underline{t_1}\ mod\ \underline{t_2}$	Modulus.
$\underline{t_1}(t_2)$	Function application.
$spread(\underline{t_1}; x, y.t_2)$	Pair analyzer.
$decide(\underline{t_1}; x.t_2; y.t_3)$	Analyzer for elements in disjoint union.
$list\_ind(\underline{t_1}; t_2; x, y, z.t_3)$	List induction.
$ind(\underline{t_1}; w, x.t_2; t_3; y, z.t_4)$	Integer induction.
$rec\_ind(\underline{t_1}; x, y.t_2)$	Induction on recursively defined elements.
$int\_eq(\underline{t_1}; \underline{t_2}; t_3; t_4)$	If-then-else form for integer equality.
$less(\underline{t_1}; \underline{t_2}; t_3; t_4)$	If-then-else form for integer ordering.

Table XIII: Noncanonical terms in Nuprl. Principal arguments are underscored.
(Subscripted) t's denote terms.
w, x, y, z denote variables.

Redex		Contractum
$-i$	$\Rightarrow$	the negation of i
$i+j$	$\Rightarrow$	the sum of i and j
$i-j$	$\Rightarrow$	the result of subtracting j from i
$i*j$	$\Rightarrow$	the result of multiplying i and j
i/j	$\Rightarrow$	the largest integer n such that $(n-1)*j < n*j \leq i$
$i \ mod \ j$	$\Rightarrow$	the result of $i - j*(i/j)$
$(\lambda x.\ t_1)(t_2)$	$\Rightarrow$	$t_1[t_2/x]$
$spread(\langle t_1, t_2\rangle; x, y.t_3)$	$\Rightarrow$	$t_3[t_1/x, t_2/y]$
$decide(inl(t_1); x.t_2; y.t_3)$	$\Rightarrow$	$t_2[t_1/x]$
$decide(inr(t_1); x.t_2; y.t_3)$	$\Rightarrow$	$t_3[t_1/x]$
$list\_ind(nil; t_3; x, y, z.t_4)$	$\Rightarrow$	t_3
$list\_ind(t_1.t_2; t_3; x, y, z.t_4)$	$\Rightarrow$	$t_4[t_1/x, t_2/y, list\_ind(t_2; t_3; x, y, z.t_4)/z]$
$ind(0; w, x.t_1; t_2; y, z.t_3)$	$\Rightarrow$	t_2
$ind(i; w, x.t_1; t_2; y, z.t_3)$	$\Rightarrow$	$t_1[i/x, ind(i+1; w, x.t_1; t_2; y, z.t_3)/y]$ if $i < 0$
$ind(i; w, x.t_1; t_2; y, z.t_3)$	$\Rightarrow$	$t_3[i/x, ind(i-1; w, x.t_1; t_2; y, z.t_3)/y]$ if $i > 0$
$rec\_ind(c; x, y.t)$	$\Rightarrow$	$t[c/x, \lambda z.\ rec\_ind(z; x, y.t/y]$
$int\_eq(i; j; t_1; t_2)$	$\Rightarrow$	t_1 if $t_1 = t_2$
$int\_eq(i; j; t_1; t_2)$	$\Rightarrow$	t_2 if $t_1 \neq t_2$
$less(i; j; t_1; t_2)$	$\Rightarrow$	t_1 if $i < j$
$less(i; j; t_1; t_2)$	$\Rightarrow$	t_2 if $i \geq j$

Table XIV: Reduction rules in Nuprl.
(Subscripted) t's denote terms.
c denotes a canonical term.
i, j denote integer constants.

References

[1] Aczel, P. "Lecture Notes on Non-Well-Founded Sets." *CSLI Lecture Notes 11*, CSLI, Stanford, 1988.

[2] 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.

[3] Cleaveland, R. "Type-Theoretic Models of Concurrency." PhD. Thesis, Cornell University, May 1987.

[4] Cleaveland, R. and P. Panangaden. "Type Theory and Concurrency." Technical Report 85-714, Department of Computer Science, Cornell University.

[5] Constable, R. et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, 1986.

[6] Curry, H., R. Feys and W. Craig. *Combinatory Logic* 1. North-Holland, Amsterdam, 1958.

[7] DeNicola, R. and M. Hennessy. "Testing Equivalences for Processes." *Theoretical Computer Science* 24, n. 1, January 1984, pp. 83-133.

[8] Hennessy, M. "Acceptance Trees." *Journal of the Association for Computing Machinery* 32, n. 4, October 1985, pp. 896-928.

[9] Hennessy, M. *An Algebraic Theory of Processes.* MIT Press, Boston, 1988.

[10] 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.

[11] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[12] 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.

[13] Howe, D. "Implementing Number Theory: An Experiment with Nuprl." *Proceedings of the Eighth International Conference on Automated Deduction*, Lecture Notes in Computer Science 230, J. Siekmann ed. Springer-Verlag, Berlin, 1986.

[14] Martin-Lof, P. "Constructive Mathematics and Computer Programming." *Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science*, pp. 153-175. North-Holland, Amsterdam, 1982.

[15] 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.

[16] Milner, R. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer-Verlag, Berlin, 1980.

[17] Milner, R. "Calculi for Synchrony and Asynchrony." *Theoretical Computer Science* 25, n. 3, July 1983, pp. 267-310.

[18] 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.

[19] Stirling, C. "Modal Logics for Communicating Systems." To appear in *Theoretical Computer Science.*