Nuprl's Class Theory and its Applications

Robert L. Constable
Cornell University

Jason Hickey
Caltech

Abstract

This article presents a theory of classes and inheritance built on top of constructive type theory. Classes are defined using dependent and very dependent function types that are found in the Nuprl constructive type theory. Inheritance is defined in terms of a general subtyping relation over the underlying types. Among the basic types is the intersection type which plays a critical role in the applications because it provides a method of composing program components.

The class theory is applied to defining algebraic structures such as monoids, groups, rings, etc. and relating them. It is also used to define communications protocols as infinite state automata. The article illustrates the role of these formal automata in defining the services of a distributed group communications system. In both applications the inheritance mechanisms allow reuse of proofs and the statement of general properties of system composition.

1 Introduction

The results presented here were created as part of a broad effort to understand how to use computers to significantly automate the design and development of software systems. This is one of the main goals of the "PRL project" at Cornell[1]. One of the basic tenants of our approach to this task is that we should seek the most naturally expressive formal language in which to specify the services, characteristics and constraints that a software system must satisfy.

If the formal expression of services is close to a natural one, then people can more readily use it. We also want to allow very compact notations for concepts used to describe systems, and this effect is also a consequence of expressive richness. We have discovered that it is frequently the case that the system we have built to implement one formal language will support an even richer one. So we have come to see our work as also progressively improving the reach of our tools.

Achieving a rich formalism is an on-going task because we are quite far from understanding the full variety of mechanisms and concepts that are used by systems designers, and we discover new concepts in the context of any formalism built to capture what we have understood well enough to formalize. This paper reports on new results in this on going process. To understand our new ideas requires explaining the starting point; we do that in the section on types.

The new ideas are based on a definition of records and dependent records that we started exploring several years ago [11, 17, 22] and is related to recent work in Martin-Löf type theory [5]. This is the subject of section 3, Classes. Class theory provides the basis for formalizing object-oriented concepts such as classes and inheritance. The way we present these notions connects in an unexpected way to the basic ideas behind Girard's new logic, Ludics [19].

The most widely accessible applications of these ideas is to the basic structures and containment among them of algebra, e.g. relating monoids, groups, rings, algebras and so forth. Indeed for us, several of the ideas arose in the investigation of symbolic algebra in Nuprl. The first public discussion of these ideas came in the first author's lectures about this work to the symbolic algebra community (1997). We will see an especially elegant presentation of the algebraic concepts that is made possible by providing a uniform treatment of naming of the fields in a structure. This idea

[1]The scope of this project is defined at it's Cornell web site www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html. This work was supported by DARPA under grant F30602-98-2-0198.

arose in the context of formalizing a software system [6], and we describe the system applications in section 5, Applications to systems.

The concepts of record and subtyping allow us to express the basic ideas of object-oriented programming. What we observe is that although it has been difficult to provide a clean mathematical foundation for all aspects of object-oriented programming (see the attempts by Cardelli [8, 20, 21]), nevertheless we can account elegantly for the concepts of *class* and *method.*

2 Types

2.1 The Core Theory

Type theory is one of the most successful foundational theories for computer science [11, 10, 9]. It is a mathematical theory which has provided solutions to a large number of the practical problems. It is an excellent basis for the semantics of functional and procedural programming languages. [2] It provides guidlines for the design of such languages, as work on ML illustrates. Type theory also guides the compilation of languages such as ML, Haskell, and Java. It provides the basis for programming logics and program verification [14, 3, 36, 15]. These solutions are not "perfect", and there is more work to be done as better programming languages are designed.

One reason for the constant evolution of programming languages is that there is a fundamental compromise between expressiveness and compilation technology. As we come to understand compilation of rich languages such as ML, we learn to handle more features efficiently, so languages can be enriched. There is an unexpected contribution to this process from type theory. Namely, these theories define an internal programming language. In the case of Nuprl this language is very much like ML with a much richer type system. So Nuprl is an "idealized programming language" in the sense that if we could efficiently compile it, we would have achieved a local maximum in language technology. So we can judge the practical languages such as ML, Java, Modula and Haskell in terms of how few idosyncratic, compilation dictated restrictions there are in the language definition. The fewer there are, the closer the language approximates one of the ideal languages. This gives designers a target and a rough metric of success.

2.2 Primitive Types

void is a type with no elements
unit is a type with one element, denoted $\bullet$

There will be other primitive types introduced later. Notice, in set theory we usually have only two primitive sets, ϕ, the empty set, and some infinite set (usually ω).

Compound Types We build new types using *type constructors.* These tell us how to construct various kinds of objects. (In pure set theory, there is only one kind, *sets*).

The type constructors we choose are motivated both by mathematical and computational considerations. So we will see a tight relationship to the notion of *type* in programming languages. The notes by C.A.R. Hoare, *Notes on Data Structuring*, make the point well [23].

2.3 Cartesian products

If A and B are types, then so is their Cartesian product, written $A \times B$. There will be many *formation rules* like this, telling us how to construct new types from existing ones; so we adopt a simple convention for stating them in the style of inference rules. We write

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A \times B \;\text{ is a } Type.}$$

[2]The type theory we have in mind is one rich enough to include the theory of *domains* which accounts for partial objects, such as partial functions.

The elements of a product type are pairs, $\langle a, b\rangle$. Specifically if a belongs to A and b belongs to B, then $\langle a, b\rangle$ belongs to $A \times B$. We abbreviate this by writing rules for type membership as inference rules.

$$\frac{a \ \in \ A \qquad b \ \in \ B}{\langle a, b\rangle \ \in \ A \times B.}$$

We say that

$$\langle a, b\rangle = \langle c, d\rangle \text{ in } A \times B \text{ iff } a = c \text{ in } A \text{ and } b = d \text{ in } B.$$

There is essentially only one way to decompose pairs. We say things like, "take the first element of the pair P," symbolically we might say $first(P)$ or $1of(P)$. We can also "take the second element of P," *second*(P) or $2of(P)$.

In programming languages these types are generalized to n-ary products, say $A_1 \times A_2 \times \ldots \times A_n$.

In set theory, equality is uniform and built-in, but in type theory we define equality with each constructor, either built-in (as in Nuprl) or by definition as in this core theory.

Function Space

We use the words "function space" as well as "function type" for historical reasons. If A and B are types, then $A \to B$ is the type of *computable functions* from A to B. These are given by rules which are defined on each a in A and which produce a *unique value.* We summarize by

$$\frac{A \text{ is a } \mathit{Type} \quad B \text{ is a } \mathit{Type}}{A \to B \ \text{ is a } \mathit{Type}.}$$

One common informal notation for functions is seen in algebra texts, e.g. Bourbaki's *Algebra* [7]; namely, we write expressions like $x \mapsto b$ or $x \overset{f}{\mapsto} b$; the latter gives a name to the function. For example, $x \mapsto x^2$ is the squaring function on numbers.

If b computes to an element of B when x has value a in A for each a, then we say $(x \mapsto b) \ \epsilon \ A \to B$. Formally we will use lambda notation, $\lambda(x.b)$ for $x \mapsto b$. The natural rule for typing a function $\lambda(x.b)$ is to say that $\lambda(x.b) \in A \to B$ provided that when x is of type A, b is of type B. We can express these *typing judgments* in the form $x : A \vdash b \in B$. The typing rule is then

$$\frac{x : A \vdash b \in B}{\vdash \lambda(x.b) \in A \to B.}$$

If f, g are functions, we define their equality as

$$f = g \quad \text{ iff } f(x) = g(x) \quad \text{for all} \quad x \text{ in } A.$$

If f is a function from A to B and $a \in A$, we write $f(a)$ for the value of the function.

2.4 Disjoint Unions (also called Discriminated Unions)

Forming the union of two sets, say $x \cup y$, is a basic operation in set theory. It is basic in type theory as well, *but* for computational purposes, we also want to discriminate based on which type an element is in. To accomplish this we put tags on the elements to keep them disjoint and define the concept of a *disjoint* union. Later we define a more general union. Here we use *inl* and *inr* as the tags.

$$\frac{A \text{ is a } Type \quad B \text{ is a } Type}{A + B \ \text{ is a } Type.}$$

The membership rules are

$$\frac{a \ \in \ A}{inl(a) \ \in \ A + B} \qquad \frac{b \ \in \ B}{inr(b) \ \in \ A + B.}$$

We say that $inl(a) = inl(a')$ iff $a = a'$ and likewise for $inr(b)$.
We can now use a case statement to detect the tags and use expressions like

$$\begin{array}{lll} \text{if } x = inl(z) & \text{then} \ldots & \text{some expression in } z \ldots \\ \text{if } x = inr(z) & \text{then} \ldots & \text{some expression in } z \ldots \end{array}$$

in defining other objects. The test for $inl(z)$ or $inr(z)$ is computable. There is an operation called *decide* that discriminates on the type tags. The typing rule and syntax for it are given in terms of a typing judgment of the form $E \vdash t \in T$ where E is a list of declarations of the form $x_1 : A_1, \ldots, x_n : A_n$ called a *typing environment*. The A_i are types and x_i are variables declared to be of type A_i. The rule is

$$\frac{E \vdash d \in A + B \quad E, u : A \vdash t_1 \in T \quad E, v : B \vdash t_2 \in T}{E \vdash decide(d; u.t_1; v.t_2) \in T.}$$

2.5 Dependent Product

Suppose we wish to construct a type representing the date:

$$\begin{array}{rcl} Month & = & \{1, \ldots, 12\} \\ Day & = & \{1, \ldots, 31\} \\ \\ Date & = & Month \times Day \end{array}$$

We would need a way to check for valid dates. The pair, $\langle 2, 31 \rangle$ is a perfectly legal member of *Date*, although it is not a valid date. One thing we can do is to define

$$\begin{array}{rcl} Day(1) & = & \{1, \ldots, 31\} \\ Day(2) & = & \{1, \ldots, 29\} \\ & \vdots & \\ Day(12) & = & \{1, \ldots, 31\} \end{array}$$

and now write our data type as a "dependent product"

$$Date = m : Month \times Day(m).$$

We mean by this that the second element of the pair belongs to the type indexed by the first element. Now, $\langle 2, 20 \rangle$ is a legal date since $20 \in Day(2)$, and $\langle 2, 31 \rangle$ is illegal because $31 \notin Day(2)$.

Many programming languages implement this or a similar concept in a limited way. An example is Pascal's *variant records*. While Pascal requires the indexing element to be of scalar type, we will allow it to be of any type.

We can see that what we are doing is making a more general type. It is very similar to $A \times B$. Let us call this type $prod(A; x.B)$. We can display this as $x : A \times B$. The typing rules are:

$$\frac{E \vdash a : A \quad E \vdash b \;\epsilon\; B[a/x]}{E \vdash pair(a,b) : prod(A;x.B).}$$

$$\frac{E \vdash p : prod(A;x.B) \quad E, u : A, v : B[u/x] \vdash t \;\epsilon\; T}{E \vdash spread(p;u,v.t) \;\epsilon\; T.}$$

Note that we haven't added any elements. We've just added some new typing rules.

2.6 Dependent Functions

If we generalize the type $A \rightarrow B$ by allowing B to be a family of types indexed by A, then we get a new type, denoted by $fun(A;x.B)$, or $x : A \rightarrow B$. The rules are:

$$\frac{E, y : A \vdash b[y/x] \in B[y/x]}{E \vdash \lambda(x.b) \in fun(A;x.B)} \; new\ y$$

$$\frac{E \vdash f \in fun(A;x.B) \quad E \vdash a \in A}{E \vdash ap(f;a) \in B[a/x].}$$

Example 2 : Back to our example *Dates*, we see that $m : Month \rightarrow Day[m]$ is just $fun(Month;m.Day)$, where Day is a family of twelve types, and $\lambda(x.maxday[x])$ is a term in it.

2.7 Universes

A critical property of our type theory is that types are also objects. The collection of all types built from the atomic types by the operations we have defined form a type called a *universe*. A universe is an example of a "large type" — a type whose members include types. Following Martin-Löf [31] we consider a hierarchy of universes, $\mathbb{U}_1, \mathbb{U}_2, \ldots$ Each $\mathbb{U}_i$ belongs to all $\mathbb{U}_j$ with $j > i$, and the hierarchy is cumulative, i.e. $\mathbb{U}_i \sqsubseteq \mathbb{U}_{i+1}$.

We treat universe indexes ambiguously, writing U_i to denote any universe. We sometimes write *Type* when the universe level is not critical. Universes are discussed extensively by Allen [2] and Constable [11]. We use them later to define records and classes.

2.8 Propositions as types

One of the key distinguishing features of Martin-Löf type theory [30, 31, 32] and the Nuprl type theory [12, 10, 11] is that propositions are considered to be types, and a proposition is true iff it is inhabited. The inhabitants of a proposition are mathematical objects that provide evidence for the truth of the proposition.

This approach to propositions is related to the so-called *Curry-Howard isomorphism* [18, 24] between propositions and types. But the correspondance is elevated in our type theory to the status of a definition. The Curry-Howard isomorphism explains why the definition is sensible for Heyting's formalization of Browwer's constructive semantics for propositions. We briefly recall these semantics and state the Nuprl definitions for the logical operators. According to Browwer the logical operators have these constructive meanings:

$\perp$	is never true (read $\perp$ as "false").
$A \;\&\; B$	is true (provable) iff we can prove A and we can prove B.
$A \vee B$	is true iff either we can prove A or we can prove B, and we say which is proved.
$A \Rightarrow B$	is true iff we can effectively transform any proof of A into a proof of B.

$\neg A$	holds iff $A \Rightarrow \perp$.
$\exists x : A.B$	is true iff we can construct an element if type A, say a, and a proof of $B[a/x]$.
$\forall x : A.B$	is true iff we have an effective method to construct a proof of $B[a/x]$ for any a in A.

For an atomic proposition, P, with no logical substructure, we say it is true exactly when we have evidence, P, for it. The evidence can be taken to be atomic as well, e.g. unstructured. So we use the unit element, $\bullet$, as evidence.

We express Browwer's semantics by these definitions of the logical operators:

Definition	$A \ \& \ B$	$==$	$A \times B$
	$A \vee B$	$==$	$A + B$
	$A \Rightarrow B$	$==$	$A \rightarrow B$
	$\exists x : A.B$	$==$	$x : A \times B$
	$\forall x : A.B$	$==$	$x : A \rightarrow B$
	$\perp$	$==$	$void$
	$\top$	$==$	$\mathbf{1}$.

We can also define other logical operators that express shades of meaning not commonly expressed. For example, we will see later that intersection, $\cap x : A.B$, makes sense as a kind of universal quantifier.

We use $Prop_i$ as a synonym for $Type_i$. We also abbreviate $Prop$ as $\mathbb{P}$.

Classical Nuprl

Doug Howe [25, 26] has shown that Nuprl, as presented here, is consistent if we add the axioms $\forall P : Prop_i.(P \vee \neg P)$. He shows how to interpret types as sets and replace computable functions with set theoretic graphs. In Howe's semantics, the definitions given above for logical operators correspond to the usual classical logical operators.

2.9 Subset Types

According to the previous section, among the objects of mathematics we consider are *propositions*. For example, $0 =_N 0$ and $0 < 1$ are true propositions about the type N. We also consider *propositional forms* such as $x =_N y$ or $x < y$. These are sometimes called *predicates*.

The type of all (small) propositions is $Prop$. *Propositional functions* on a type A are elements of the type $A \rightarrow Prop$.

We also need types that can be restricted by predicates such as $\{x : N | x = 0 \text{ or } x = 1\}$. This type behaves like the *Booleans*.

The general formation rule for subset types is this. If A is a type and $B : A \rightarrow Prop$, then $\{x : A | B(x)\}$ is a *subtype* of A. The elements of $\{x : A | B(x)\}$ are those a in A for which $B(a)$ is true. Sometimes we state the formation in terms of predicates, so if P is a proposition for any x in A, then $\{x : A | P\}$ is a subtype of A.

2.10 Intersection types

Given two types A and B, it makes sense to consider the elements they have in common; we call that type the *intersection* of A and B, written $A \cap B$. We require that ($a = b$ in $A \cap B$) iff $a = b$ in A and $a = b$ in B. For example, it is clear that $void \cap A$ is $void$ for any type A and $A \cap A$ is A.

It might be a suprise that $(1 \rightarrow 1) \cap (void \rightarrow void)$ is not $void$ but contains the identity function, $\lambda(x.x)$. This is because the base objects of Nuprl are *untyped*, so $\lambda(x.x)$ is polymorphic, indeed belonging to all types $A \rightarrow A$. It is clear that $\{x : A | P(x)\} \cap \{x : A | Q(x)\}$ is $\{x : A | P(x) \ \& \ Q(x)\}$.

The intersection type is defined for a family of types as well. Let $B(x)$ be a family of types for $x \epsilon A$, then $\cap \ x : A.B(x)$ is their intersection; and $b \ \epsilon \ \cap \ x : A.B(x)$ iff $b \ \epsilon \ B(a)$ for all $a \ \epsilon \ A$. also

$b = b'$ in $\cap\, x : A.B(x)$ iff $b = b'$ in $B(a)$ for all $a \epsilon A$. Notice that when A is empty, $\cap\, x : A.B(x)$ is not empty but has exactly one element. This element is denoted by any closed expression of the theory, e.g. $void = 1$ in $\cap\, x : void.B(x)$. Such equalities follow from the understanding that under the asumption that $x \epsilon void$, we can infer anything.

Types such as $\cap\, x : void.B(x)$ are maximal in the sense that every type is a subtype and every closed term denotes the single member of the type. We pick one such type expression and call it the *top type*, denoted Top. For example, $\cap\, x : void.x$ can be Top.

Type equality follows the pattern for structural equality, namely

$$(\cap\, x : A.B(x) = \cap\, x : A'.B'(x)) \text{ iff } A = A' \text{ and } B(a) = B'(a).$$

Intersection types can express conditional typehood, which is useful in typing partial functions. If B does not depend on x, then $\cap\, x : A.B$ expresses the notion that *B is a type provided that A is inhabited*. We write this as B *given* A, following an analysis of this idea by Stuart Allen. Allen calls these *guarded types.*

Guarded types can specify the conditions under which $A \to B$ is a type in Nuprl. Namely, if A is nonempty, then B must be a type. So the typing is

$$\lambda(A, B.A \to B) \ \in \ (A : Type \to B : (\cap\, x : A.Type) \to Type).$$

Guarded types are also useful in expressing one-one correspondances between types. For example we know that mappings from pairs p in $x : A \times B(x)$ into $C(p)$ are isomorphic to curried mappings of the type $x : A \to y : B(x) \to C(\langle x, y\rangle)$. Similarly, mappings $x : \{y : A \mid B(y)\} \to C(x)$ are isomorphic to those in $x : A \to \cap\, y : B(x).C(x)$.

2.11 Subtyping

There is a natural notion of subtyping in this theory. We say that $A \ \sqsubseteq\ B$ iff $a = a'$ in A implies that $a = a'$ in B. For example, $\{x : A \mid P(x)\} \ \sqsubseteq\ A$ for any predicate $P(x)$. We clearly have $void \ \sqsubseteq\ A$ for any type A and $A \ \sqsubseteq\ Top$ for any type A, and $A \cap B \sqsubseteq A$ for any types A and B. It is easy to see that the following relationships hold.

	$A \sqsubseteq A'$	$B \sqsubseteq B'$	
1.	$A \times B$	$\sqsubseteq$	$A' \times B'$
2.	$A + B$	$\sqsubseteq$	$A' + B'$
3.	$A' \to B$	$\sqsubseteq$	$A \to B'$

The relation in 3 holds because functions in type theory are polymorphic. Given $f \epsilon A' \to B$, we see that it belongs to $A \to B'$ as well because on inputs *restricted to* A it will produce results in B. In set theory this relationship would not hold.

3 Classes

3.1 Records

In the context of the type theory in section 2, there is more than one natural idea of a "class". We could borrow the idea from set theory that a class is a "large collection", one too large to be considered a set (see for example Bernays [4]). In our theory this corresponds to taking classes as "large types", i.e. types which contain types as members, in proper elements of universe, U_2.

We follow instead the idea from algebra and programming languages that classes are like algebraic structures, they are sets with operators. But classes are defined using signatures, and these are large types. Without the signature definition, as a large type, we could not have the definition of a class as a small type. The precise type we start with is a record type, these will also be classes.

Let F be a discrete type, the field names, and define a signature over F as a function from F into types, say $Sig \ \in\ F \to$ Types.

Definition: a *record type on discrete type F with signature $Sig \in F \to$* Types is the dependent type $i : F \to Sig(i)$.

Notation: If F is a finite type, say $\{x_1, \ \ldots \ , x_n\}$, then we write the record as $\{x_1 : Sig(x_1); \ \ldots \ ; x_n : Sig(x_n)\}$. If we are given types $T_1, \ \ldots \ , T_n$ we can also write simply $\{x_1 : T_1; \ \ldots \ ; x_n : T_n\}$ and this implies that we built the function assigning type T_i to name x_i.

Notice that if F_1 and F_2 are discrete types such that $F_1 \sqsubseteq F_2$ and
$Sig_1 : F_1 \to$ Type, $Sig_2 : F_2 \to$ Type and $Sig_1(x) \sqsubseteq Sig_2(x)$ for $x \epsilon F_1$ then

$$i : F_2 \to Sig_2(i) \quad \sqsubseteq \quad j : F_1 \to Sig_2(j).$$

For example, if $F_1 = \{x_1, y\}$ and $F_2 = \{x, y, z\}$ then $\{x_1 : T_1; y : T_2; z : T_3\} \quad \sqsubseteq \quad \{x : T_1; y : T_2\}$, where $Sig_1(x) = T_1$, $Sig_1(y) = T_2$, $Sig_2(x) = T_1$, $Sig_2(y) = T_2, Sig_2(3) = T_3$. In this case $Sig_1(i) = Sig_2(i)$ for $i \epsilon \{x, y\}$, but we could allow $\{x : T_1'; y : T_2'\}$ where $T_1' \sqsubseteq T_1, T_2' \sqsubseteq T_2$.

Consider $\{0,1\} = N_2 \subseteq N_3 = \{0,1,2\}$, then $N_3 \to B \sqsubseteq N_2 \to B$. For instance, consider f defined by this table,

x	$f(x)$
0	*tt*
1	*tt*
2	*ff*

Then, $f \ \epsilon \ N_3 \to B$ implies $f \ \epsilon \ N_2 \to B$ by ignoring arguments, e.g. ignore $f(2)$. In set theory $\{\langle 0, tt\rangle, \langle 1, tt\rangle, \langle 2, ff\rangle\}$ is clearly not a subset of $\{\langle 0, tt\rangle, \langle 1, tt\rangle\}$. If we used explicitly typed function terms, such as $\lambda x^A.b$, then $\lambda x^{N3}.b \ \not\epsilon \ N_2 \to b$ since $N_3 \not\subseteq N_2$.

An important specific example of these relationships will be discussed extensively in section 4. We consider it here briefly. Given a type M as the carrier of algebraic structure, we can define the structure of a monoid over M, $Monoid_M$, as $\{op : M \times M \to M; id : M\}$. By adding a component, we can extend this to a group structure, $Group_M$, as $\{op : M \times M \to M; op : M; inv : M \to M\}$. Notice that $Group_M \sqsubseteq Monoid_M$.

A monoid over a type M is a pair $\langle op, id\rangle$ where $op \ \epsilon \ M \times M \to M$ is associative and $id \ \epsilon \ M$ is an identity. A group over M is $\langle op, id, inv\rangle$ where $\langle op, id\rangle$ is a monoid and $inv \in M \to M$ is an inverse.

3.2 Uniform Records

Extending records is a basic operation, as in extending a Monoid structure to a Group structure. This can be done in a uniform way if we agree on a common set of names to be used as names of the components ("fields" as they are often called). Let us take *Label* as an infinite discrete type. Signatures *Sig* over *Label* are functions in $i : Label \to Type$, and records with this signature are $i : Label \to Sig(i)$. We call these *uniform records*, and also denote them $\{Sig\}$.

We generally consider records in which only finitely many labels are names of significant components. If we map the insignificant components to the type *Top*, then intersection of arbitrary uniform records makes sense and is defined as

$$(i : Label \to Sig_1(i)) \ \cap \ (j : Label \to Sig_2(j)) \ = i : Label \to Sig_1(i) \ \cap \ Sig_2(i).$$

Record intersection is an operation that extends records as we see if we assume that *op*, *id* and *inv* are elements of *Label* and assume that $MonSig_M(i) = Top$ if $i \neq op$ and $i \neq id$. Generally when we write $\{x_1 : Sig(x_1); \ \ldots \ ; x_n : Sig(x_n)\}$ we will assume that $Sig(i) = Top$ for $i \neq x_j$. So now consider $\{op : M \times M \to M; id : M\} \ \cap \ \{inv : M \to M\}$.

The result is $Group_M$, $\{op : M \times M \to M; id : M; inv : M \to M\}$ because in the structure $\{inv : M \to M\}$, the components for *op* and *id* are *Top* and in Monoid the component for *inv* is *Top*. Thus for example, the type at *op* in the intersected structure is

$$op : (M \times M \to M) \ \cap \ Top \ \text{ and } \ (M \times M \to M) \ \cap \ Top = M \times M \to M.$$

When two records share a common field, the intersected record has the intersection of the types, that is $\{x : T\} \cap \{x : S\} = \{x : T \cap S\}$ So generally,

$$\{x : T\} \cap \{y : S\} = \begin{cases} \{x : T, y : S\} & if\ x \neq y \\ \{x : T \cap S\} & if\ x = y \end{cases}$$

3.3 Dependent Records

Records can be seen as (Cartesian) products with labels for the components. Can we define the record analogue of dependent products? The right notation would be $\{x_1 : T_1; x_2 : T_2(x_1);\ \ldots\ ; x_n : T_n(x_1,\ \ldots\ , x_{n-1})\}$ where $T_i(x_1,\ \ldots\ , x_{i-1})$ is a type depending on $x_1,\ \ldots\ , x_{i-1}$.

Can we define this structure in terms of a function $Sig : \{x_1,\ \ldots\ , x_n\} \rightarrow Type$? We see that $Sig(x_1)$ is just a type, but $Sig(x_2)$ uses a function $F \in Sig(x_1) \rightarrow Type$, and it is F applied to x_1. We might think of writing $F(x_1)$, but x_1 is a name and F wants an element of $Sig(x_1)$. Basically we need to talk about an arbitrary element of the type we are building. That is if $r \in i : \{x_1,\ \ldots\ , x_n\} \rightarrow Sig(i)$, then $Sig(x_2)$ is $F(r(x_1))$. So to define Sig we need r, and to type r we need Sig.

We see that an element of a type such as $\{x_1 : T_1;\ \ldots\ ; x_n : T_n(x_1,\ \ldots\ , x_{n-1})\}$ has the property that its type depends on its "previous values". That is, the type of $r(x_1)$ is $Sig(x_1)$, but the type of $r(x_2)$ is $Sig(x_2)(r(x_1))$.

We can encode this type using the *very-dependent function type* denoted $\{f \mid x : A \rightarrow B[f, x]\}$. We define it below. Now we use it to define the dependent records. We take A to be the finite type $\{0 \ldots (n-1)\}$. Let

$$T_f(n) \equiv \left\{ f \;\middle|\; i\!: \{0 \ldots (n-1)\} \rightarrow \begin{bmatrix} case(i)\, of & & \\ 0 & \Rightarrow & T_0 \\ 1 & \Rightarrow & T_1(f\,(0)) \\ & \vdots & \\ n-1 & \Rightarrow & T_{n-1}(f\,(0))(f\,(1)) \cdots (f\,(n-2)) \end{bmatrix} \right\}.$$

If we have a function F that returns the values T_i for $0 \leq i < n$ given the index i, we can give this type a more concise form. Let $fix(f.b)$ define the fixed point of the program b. That is, $fix(f.b) \equiv Y \lambda f.b$. Given functions F and f, and an index i we can define a function *napply* that applies F to $f(0), \ldots, f(n-1)$:

$$napply \equiv \lambda f.\lambda i.fix(g.\lambda G.\lambda j.\ \textbf{if}\ j = i\ \textbf{then}\ G\ \textbf{else}\ g(G(f(j)))(j+1)).$$

The arguments to *napply* are as follows:

i is an integer that is the index of the final argument,

j is an integer that is the index of the first argument,

G is the value that will be applied to the arguments $f(j), f(j+1), \ldots, f(i-1)$,

f is a function returning the arguments.

Computing *napply* gives

$$napply\ (f)(i)(G)(j) \equiv G(f(j))(f(j+1)) \cdots (f(i-1)).$$

Using *napply*, the type $T_f(n)$ is then described as:

$$T_f(n) \equiv \{f | i : \{0 \ldots (n-1)\} \rightarrow napply(f)(i)(F(i))(0)\}.$$

The type function F can be defined in a similar manner. Given F, and an index i, we can compute the i-ary dependent function type with the term:

$$
\begin{array}{l}
nfun \equiv \lambda F.\lambda i.fix(g.\lambda h.\lambda j.\ \textbf{if}\ \ j = i\ \textbf{then}\ \mathbb{U}_i \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ x : napply\ (h)(j)(F(j))(0) \rightarrow g(\lambda k.\ \textbf{if}\ k = j\ \textbf{then}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ h(k))(j+1).
\end{array}
$$

The arguments are:

i is an integer that is the index of the last type in the function type being constructed,

j is an integer that is the index of the first type in the functin type being constructed,

F is the value that will be applied to get the argument type of the function type being constructed,

h is a function defined on $\{0 \ldots (j-1)\}$ that returns the initial arguments of the dependent function type being constructed.

After computation, we have

$$
\begin{array}{rcl}
nfun(F)(i)(h)(j) & \equiv & x_j : F(j)(h(0))(h(1)) \ldots (h(j-1)) \\
& \rightarrow & x_{j+1} : F(j+1)(h(0))(h(1)) \ldots (h(j-1))(x_j) \\
& \vdots & \\
& \rightarrow & x_{i-1} : F(i-1)(h(0))(h(1)) \ldots (h(j-1))(x_j) \ldots (x_{i-2}) \\
& \rightarrow & \mathbb{U}_i.
\end{array}
$$

In the normal case, $j = 0$ and the argument h is not used, and we have

$$
\begin{array}{rcl}
nfun(F)(i)(h)(0) & \equiv & x_0 : F(0) \\
& \rightarrow & x_1 : F(1)(x_0) \\
& \vdots & \\
& \rightarrow & x_{i-1} : F(i-1)(x_0)(x_1) \cdots (x_{i-2}) \\
& \rightarrow & \mathbb{U}_i.
\end{array}
$$

The type for F is then:

$$
T_F(n) \equiv \{F | i : \{0 \ldots (n-1)\} \rightarrow nfun(F)(i)(\lambda x.x)(0)\}.
$$

If we parameterize these types over n, we have the types

$$
\begin{array}{c}
S_f \equiv n : \mathbb{N} \times T_f(n) \\
S_F \equiv n : \mathbb{N} \times T_F(n).
\end{array}
$$

The type S_F specifies *descriptions* of dependent products of arbitrary finite arity, and the type S_f specifies the corresponding dependent product type. The inhabitants of S_f are pairs $\langle n, f \rangle$ where n is the arity of the product, and f is a function with domain $\{0 \ldots (n-1)\}$ that serves as the projection function.

This finite arity dependent product type will be useful to define the type *Signature*. It can also be used to define the type of sequents in a constructive logic: the hypotheses of the sequents are defined as a dependent product of finite arity, and the goal is a type that may depend on all hypotheses. We will cover next a more formal treatment of the semantics of the function type.

3.4 Semantics of the very-dependent function type

Next we will discuss briefly a semantics for the very-dependent function type, restricted to non-cyclic definitions. There are interesting variations on the semantics that allow cyclic definitions, or definitions that derive the order from the computation in the range, or that allow the expression of partial function types. However, these semantics are an area of research—they do not fit in well with the current semantics of the type theory, and it seems likely that the semantics of the entire type theory would have to be modified to accomodate them. We don't discuss the extended semantics in this paper.

Simple semantics The simplest interpretation for the very-dependent function type $\{f|x : A \to B\}$ requires that there be no cycles in the type definition. That is, the term $B[f, i]$ must not evaluate f on i for any $i \in A$, and it must have a normal form. The simplest way to enforce this property is to require that A be well-ordered according to some relation $<: A \to A \to \mathbb{U}_i$, and to require that $B[f, i]$ be well-defined for f with domain $\{a : A \mid a < i\}$. This is a conservative restriction, since only the subtype of A that is used in defining the range need be well-ordered.

With this restriction, we can give a semantics for the type $\{f|x : A \to B\}$ as follows.

Semantics 1

The term $\{f|x : A \to B\}$ is a type with membership φ, if and only if:

1. A is a type with membership α,
2. A is well-founded with respect to some partial order $<$,
3. for any a such that $\alpha(a)$, there is a $\gamma_{a'}$ for all a' where $\alpha(a')$ and $a' < a$ and the following hold:
   (a) $\{f|x : \{a'' : A|a'' < a'\} \to B\}$ is a type with membership $\gamma_{a'}$,
   (b) $B[g; a']$ is a type with membership β_g for any g where $\gamma_{a'}(g)$,
   (c) for any f, $\varphi(f)$ if and only if, for all a where $\alpha(a)$ holds, $\gamma_a(f)$ and $\beta_f(f(a))$ also hold.

This semantics is well-founded because A is well-founded. The membership predicates γ and β can be constructed by induction. In the base case, the type $\{f|x : \{a'' : A|a'' < a'\} \to B\}$ reduces to $\{f|x : void \to B\}$ which is a type for any term B. Similarly, the semantics requires that at the base case that $B[g, a']$ be a type for the function g with domain *void*, which means that B cannot evaluate g on any argument. The definitions for γ and β are constructed mutually recursively for the entire domain A.

A. Kopylov has recently shown that we can define the dependent records in terms of a simple dependent intersection type $\cap\, x : A.B$ with a simple semantics.

4 Applications to Algebra

4.1 Skeletal Structures

We have seen that records parameterized by a type, the *carrier*, can represent skeletal algebraic structures. We have seen

$$Group_M \sqsubseteq Monoid_M.$$

With judicious choice of names we can add more structure, for instance to define a ring over M, we extend the Group with more operators and constants, say

$$Ring_M == \{op : M \times M \to M; id : M; inv : M \to M; op_2 : M \times M \to M; id_2 : M\}.$$

With this choice of names we have

$$Ring_M \sqsubseteq Group_M \sqsubseteq Monoid_M.$$

But we are not able to express yet that op_2 and id_2 form a second monoid. We would like $\{op_2 : M \times M \rightarrow M; id_2 : M\}$ to be a monoid and inherit properties about monoids. We will see that this can be done over uniform records by first relabeling the fields. We will have that

$$\{op_2 : M \times M \rightarrow M; id_2 : M\} \circ L \sqsubseteq Monoid_M.$$

4.2 Relabeling Uniform Structures

If $\{op_2 : M \times M \rightarrow M; id_2 : M\}$ is a uniform record over Label, then we can think of labelling the fields by composing a function $Label \rightarrow Label$ with the record function.

For example, if $L \in Label \rightarrow Label$ is defined by $L(op) = op_2, L(id) = id_2$, and if $r \in \{op_2 : M \times M \rightarrow M; id_2 : M\}$, then $r \circ L \in \{op : M \times M \rightarrow M; id : M\}$; so $r \circ L$ is an element of $Monoid_M$.

Let us define $\{op_2 : M \times M \rightarrow M; id_2 : M\} \circ L$ as $r \in \{f : (i : Label \rightarrow MonSig_M(i)) | \exists g : \{op_2 : M \times M \rightarrow M; id_2 : M\}. f = g \circ L\}$.

4.3 Full Algebraic Structure

The skeletal algebraic structures do not specify properties of the operations. To define a monoid we need to say that *op* is an associative binary operator and that *id* is an identity. Here are propositions that express these properties. We display *op* as an infix operator.

$$Assoc(M, op) == \forall x, y, z : M.((x \; op \; y) \; op \; z) = (x \; op(y \; op \; z) \text{ in } M),$$

$$Id(M, op) == \forall x : M.((id \; op \; x) = x \text{ in } M) \& ((x \; op \; id) = x \text{ in } M).$$

The *full algebraic structure* for a monoid is

$$Monoid_M == \{op : M \times M \rightarrow M; id : M; assoc : Assoc(M, op); id\_prop : Id(M, id)\}.$$

To define a group we add

$$Inv(M, op, id, inv) == \forall x : M \; ((x \; op \; inv(x)) = id \text{ in } M) \; \& ((inv(x) \; op \; x) = id \text{ in } M).$$

$$Group_M == Monoid_M \cap \{inv : M \rightarrow M; inv\_prop : Inv(M, op, id, inv)\}.$$

In this notation, we can assume that any labels that are not explicitly bound (call them free), such as *op* and *id* in the second record, are assigned the type Top.

We can continue to add properties by intersection. For example, to define *commutative (or Abelian) groups*, we first define

$$Comm(M, op) == \forall x, y : M.(x \; op \; y) = (y \; op \; x) \text{ in } M, \quad \text{then}$$

$$AbelianGroup_M == Group_M \cap \{comm : Comm(M, op)\}.$$

4.4 Colimits and Categorical Operations

There are categorical constructions, such as colimits, that can be used to unite theories. The Designware and Specware systems developed by Kestrel Inc are based on these concepts [34, 35]. We can mimic these constructions using our record calculus. Here is how we treat the basic example from the articles written by Doug Smith.

Consider two structures over E with a binary relation (decidable), one transitive and one reflexive,

$$\begin{array}{lll}
\text{say } \ SR & == & \{r_1 : E \times E \to \mathbb{B}; refl : Reflexive(E, r_1)\}, \\
\text{and } \ ST & == & \{r_2 : E \times E \to \mathbb{B}; trans : Transitive(E, r_2)\}, \\
\text{where } \ Reflexive(E, r) & == & \forall x : E.(x \ r \ x), \\
\text{and } \ Transitive(E, r) & == & \forall x, y, z : E.((x \ r \ y \ \& \ y \ r \ z) \Rightarrow x \ r \ z).
\end{array}$$

In Specware, to write these theories, the binary relations are related to each other by defining morphisms from a common skeleton as follows:

$$\{r : E \times E \to \mathbb{B}\} \xrightarrow{f} \{r_1 : E \times E \to \mathbb{B}; refl : Reflexive(E, r_1)\}$$

$$\{r : E \times E \to \mathbb{B}\} \xrightarrow{g} \{r_2 : E \times E \to \mathbb{B}; trans : Transitive(E, r_2)\}.$$

Then a colimit, SRT, with respect to f and g is defined. It has the "universal property", and these components:

$$SRT == \{rr : E \times E \to \mathbb{B}; refl : Reflexive(E, rr); trans : Transitive(E, rr)\}.$$

If we let S be the skeletal structure, then the diagram is:

$$\begin{tikzcd}
S \arrow[r, "f"] \arrow[d, "g"'] & SR \arrow[d] \\
ST \arrow[r] & SRT
\end{tikzcd}$$

In our approach, we obtain the colimit as

$$SR \circ L_1 \ \cap \ ST \circ L_2,$$

where L_1 maps rr to r_1 and L_2 maps rr to r_2.

5 Applications to Systems

5.1 Labeled Transition Systems and IO Automata

The class concept and record calculus can be used to formally describe software systems by composition from components. We illustrate this by describing the service specifications for the Ensemble group communication systems [27]. These specifications are based on IO automata - a formalism extensively studied by Lynch and Tuttle [29]. We start with a formal specification of these automata as non-deterministic transition systems.

General IO automata are instances of *labeled transition systems*. An automaton consists of a set of states Q; a set of labels A, called *actions*, that label each of the transitions of the automaton; a set of *initial* states i; and a set of allowable transitions t. In type theory, an automaton is described by the following type:

$$\begin{array}{rcl} Auto & \equiv & Q\colon Type \\ & \times & A\colon Type \\ & \times & i\colon Q \to \mathbb{P} \\ & \times & t\colon (Q \times A \times Q) \to Prop. \end{array} \tag{1}$$

This is similar to the definition used in our formalization of automata theory [13].

The initial states i are a subset of the possible states Q, defined using a predicate $Q \to \mathbb{P}$, where $\mathbb{P}$ is the set of propositions (for simplicity, this can be thought of as a Boolean value). The transition definition is similar. Note that the states and actions may be types with an infinite number of elements.

5.2 Notations for IO Automata

IO automata are often defined using stylized pseudo-code. The *actions* correspond to "significant" external events of the system being specified. For example, a simple specification of a FIFO protocol is shown below. The FIFO actions are $\text{SEND}(m)$ and $\text{RECV}(m)$, where m ranges over some message type $\mathcal{M}$. The automaton contains two queues called *sent* and *received* that save the messages. Sending a message corresponds to following a transition labeled $\text{SEND}(m)$, the message is added to the *sent* list of messages. Receiving messages corresponds to following a transition labeled $\text{RECV}(m)$. A message m can be received only when there are more messages in the *sent* queue than the *received* queue, and the next message in *sent* is m (this is a *precondition* of the transition); the effect is to add the message to the *received* queue.

FIFO	
Actions:	$\text{SEND}(m)$, $\text{RECV}(m)$, for $m \in \mathcal{M}$
State:	$sent \in \mathcal{M}$ *List*, initially empty,
	$received \in \mathcal{M}$ *List*, initially empty
$\text{SEND}(m)$	
Eff: append m to *sent*	
$\text{RECV}(m)$	
Pre: $\lvert received\rvert < \lvert sent\rvert$	
$\quad sent[\lvert received\rvert - 1] = m$	
Eff: append m to *received*	

An automaton defines a set of *executions* $q_0, a_0, q_1, a_1, \ldots$, where each triple q_i, a_i, q_{i+1} is a valid transition of the automaton. It also defines a set of *traces*, which are the actions of the valid executions. If A and B are automata, and the traces of A are a subset of the traces of B, we say that A *implements* B. An *invariant* is a predicate that is true of all states in all executions of an automaton. The FIFO automaton defines that invariant that the *received* queue is always a prefix of the *sent* queue. We abbreviate this using the satisfaction relation of the form $A \models P$ where A is an automaton and P is a safety property on traces. In this case we write, $\text{FIFO} \models \textit{always received} \leq \textit{sent}.$

5.3 Composition of IO Automata

So far, this presentation of IO automata adheres to the original form of Lynch and Tuttle.[3] Next we wish to define a form of composition that preserves safety properties, that is state invariants and properties of individual traces of the automaton. Composition of automata is discussed extensively by Lynch [28], and she states a number of theorems that relate the traces of a composition to the traces of the components. Composition identifies actions with the same name in different

[3]Lynch and Tuttle also labels actions as being *input*, *output*, or *internal* to give a little more semantic content, but we find this labeling unnecessary.

component automata, so when any one component takes this action, so do all components with this name in their signature. The traces of a composition should be the intersection of the traces of the components. Basically the composition of two automata is their product. We define it precisely now. The obvious choice for composition is the *intersection* of their traces. If M and M' are automata, their intersection has states that are states both of M and M', and traces that belong both to M and M'. If $M = (Q, A, i, t)$ and $M' = (Q', A', i', t')$, we define their intersection as follows:

$$\begin{array}{rcll} M \cap M' & \equiv & (& Q \cap Q', \\ & & & A \cap A', \\ & & & \lambda s.(i(s) \wedge i'(s)), \\ & & & \lambda q_1, a, q_2.t(q_1, a, q_2) \wedge t'(q_1, a, q_2)). \end{array} \tag{2}$$

With this definition, we have the following theorem, proved formally in Nuprl.

Theorem 1 *For any automata M and M', and state predicate P, if $M \models$ always P, then $M \cap M' \models$ always P.*

The proof is straightforward, since any execution of $M \cap M'$ is also an execution of M. In the degenerate case, if the state spaces Q and Q' do not have any elements in common, the intersection $M \cap M'$ is the empty automaton; it has no executions or traces. In our framework, we address this problem by including “all” state variables and actions in every automata, but leaving almost all of them unspecified, as shown in the following section.

The pseudo-code suggests a *style* of automata definition that can be used to guide their construction. In the pseudo-code, the actions have a *label* (like SEND or RECV), and a *value* (like $m \in \mathcal{M}$) that is associated with the label. Given a type of labels *Label*, the set of stylized actions can be defined as the dependent sum $\mathit{Action} == l : \mathit{Label}. \times F_A(l)$ where $F_A \in \mathit{Label} \rightarrow \mathit{Type}$ is a type-function that defines the type of the value associated with each label. For example, the actions of the FIFO automaton are defined by the following product:

$$l\colon \mathit{Label}. \times (\textbf{if}\, l = \text{“SEND”} \vee l = \text{“RECV”}\ \textbf{then}\ \mathcal{M}\ \textbf{else}\ \mathit{Top}).$$

If the action label is SEND or RECV, the action *value* belongs to the type of messages $\mathcal{M}$. Otherwise, the data part is totally unspecified.

The stylized states are defined similarly. A state has a set of named variables (we can use the *Label* type), and each variable has a value. The most natural type is the record type. For example, the FIFO state can be represented by the record $\{\mathit{sent} : \mathcal{M}\ \mathit{List}\,;\ \mathit{received} : \mathcal{M}\ \mathit{List}\}$.
$\mathit{State} = l : \mathit{Label} \rightarrow F_S(l)$ for some type-function $F_S \in \mathit{Label} \rightarrow \mathit{Type}$. The FIFO state has the following definition:

$$l : \mathit{Label} \rightarrow (\textbf{if}\, l = \text{“}\mathit{sent}\text{”} \vee l = \text{“}\mathit{received}\text{”}\ \textbf{then}\ \mathcal{M}\ \mathit{List}\, \textbf{else}\ \mathit{Top}).$$

Again, the state is *unspecified* on labels that are not in $\{\text{“}\mathit{sent}\text{”}, \text{“}\mathit{received}\text{”}\}$. This results in the main type-theoretic principle we use to avoid degeneracy: record *intersection* corresponds to record *concatenation*.

$$\{l_1\colon T_1; l_2\colon T_{2a}\} \cap \{l_2\colon T_{2b}; l_3\colon T_3\} \equiv \{l_1\colon T_1; l_2\colon T_{2a} \cap T_{2b}; l_3\colon T_3\}.$$

Intersection for the product type also has the concatenation property, ie., when two automata are intersected, their state variables and action signatures are concatenated. This simple feature captures the object-oriented nature of our framework: the intersection of automata corresponds to *inheritance* in object-oriented languages.

The pseudo-code can be represented in the type-theory by separating it into the four parts of an automaton. The actions define a product type, the state variables define a record (a function type), and the initial values of the state variables specify the initial conditions. The transition definitions define a transition relation. For an automata with state type Q, the transition definition $L(x)$ `Pre` $P(q,x)$ `Eff:` $\mathit{expr}(q,x)$ for $x \in T$ defines a conjunctive clause in the transition definition as follows:

$$\forall x \in T.(a = L(x) \Rightarrow P(q_1, x) \wedge (q_2 = \mathit{expr}(q_1, x) \in Q)).$$

That is, the transition (q_1, a, q_2) is in the transition relation only if, whenever the action a is $L(x)$ for some $x \in T$, the precondition P holds on the initial state q_1, and the result of the effect is equal to the final state q_2 *on the variables specified in the state* Q. Other variables are unconstrained. The `Eff:` clause of a transition definition is optional; if it is omitted, the equality condition on q_2 is omitted in the type-theoretic interpretation, allowing the state to change arbitrarily. The formal definition of the FIFO automaton is the following program:

$$\begin{array}{lrll}
Q = l : \mathit{Label} \to (\textbf{if } l = \text{“}\mathit{sent}\text{”} \vee l = \text{“}\mathit{received}\text{”} \textbf{ then } \mathcal{M}\ \mathit{List} \textbf{ else } \mathit{Top}), & & & \\
A = l.\mathit{Label} \times (\textbf{if } l = \text{“SEND”} \vee l = \text{“RECV”} \textbf{ then } \mathcal{M} \textbf{ else } \mathit{Top}), & & & \\
i = \lambda q.(q.\mathit{sent} = [] \wedge q.\mathit{received} = []), & & & \\
t = \lambda q_1, a, q_2. & \forall m \in \mathcal{M}.(a = \text{SEND}(m) & \Rightarrow & (q_2.\mathit{sent} = m \text{ appended to } q_1.\mathit{sent} \\
 & & \wedge & q_2.\mathit{received} = q_1.\mathit{received})) \\
 & \wedge \quad \forall m \in \mathcal{M}.(a = \text{RECV}(m) & \Rightarrow & (|q_1.\mathit{received}| < |q_1.\mathit{sent}| \\
 & & \wedge & q_1.\mathit{sent}[|q_1.\mathit{received}| - 1] = m \\
 & & \wedge & q_2.\mathit{received} = m \text{ appended to } q_1.\mathit{received} \\
 & & \wedge & q_2.\mathit{sent} = q_1.\mathit{sent})).
\end{array}$$

There are some drawbacks to the automata we have just presented. First, every state machine shares the same state space—in essence, all state variables are global. We use this feature during composition (like the FINITE_FIFO example), but there is no notion of "public" or "private" variables, where private variables do not become shared during composition. Second, the action space is global as well, so it is not possible to re-use an automaton in multiple specifications with different action namings.

5.4 Uniform Automata and Relabeling

Both of these problems can be addressed by introducing *renaming* operations that we defined on uniform records. We use the following informal notation, given an arbitrary renaming function $R \in \mathit{Label} \to \mathit{Label}$ on *labels*, and an automaton M:

$$\begin{array}{l}
M\text{, renaming actions } l(x) \text{ to } R(l)(x) \\
M\text{, renaming variables } v \text{ to } R(v).
\end{array}$$

When actions are renamed, the action type and the transition definitions must be modified. Consider the following stylized machine M:

$$M = (Q = l : \mathit{Label} \to F_s(l), A = l : \mathit{Label} \times F_A(l, i, t)).$$

The action renaming "M*, renaming actions* $L(x)$ *to* $R(L)(x)$" defines the renamed machine M' shown below.

$$\begin{array}{lllll}
M' = (& Q' & = & l : \mathit{Label} \to F_s(l) & \\
 & A' & = & l : \mathit{Label} \times \left(\bigcap_{\{l' \in \mathit{Label} \mid R(l') = l\}} .F_A(l')\right) & \\
 & i' & = & i, & \\
 & t' & = & \lambda q_1, l(x), q_2. \textbf{ if} & \exists l' \in \mathit{Label}.(R(l') = l) \textbf{ then} \\
 & & & & t(q_1, l'(x), q_2) \\
 & & & \qquad\qquad\quad \textbf{else} & \\
 & & & & q_1 = q_2 \in Q').
\end{array}$$

Since R is an arbitrary renaming function, it may map several labels to a single value. The action type associated with label l in A' is the *intersection* of all the action types $F_A(l')$ for any label

l' in the inverse image $R^{-1}(l)$. If $R^{-1}(l)$ is nonempty, the transition definition allows *any* of the transitions $l'(x)$ for any $l' \in R^{-1}(l)$. Otherwise, the value type on label l in A' degenerates to *Top*, and a transition is allowed only if it leaves the state unchanged. From this definition, we get the following formal meta-theorem for safety properties P:

Theorem 2 *For any automaton M, renaming function R, and state predicate P, if $M \models$ always P then $(M$, renaming actions $l(x)$ to $R(l)(x)) \models$ always P.*

The state renaming operation is similar. The states, initial state predicate, and transition relation must be modified. The following machine defines the renamed machine "$M'' = m$, *renaming variables* v *to* $R(v)$":

$$\begin{array}{rcrcl} M'' & = & (\quad Q'' & = & l : \mathit{Label} \rightarrow \left(\bigcap_{\{l' \in \mathit{Label} \mid R(l') = l\}} . F_S(l') \right), \\ & & A'' & = & l : \mathit{Label} \times F_A(l, i, t) \\ & & i'' & = & (i \circ R) \\ & & t'' & = & \lambda q_1, a, q_2 . t(q_1 \circ R, a, q_2 \circ R)). \end{array}$$

This definition induces a renaming operation on state predicates P, providing the following formal meta-theorem:

Theorem 3 *For any automaton M, renaming function R, and state predicate P, if $M \models$ always P, then $(M$, renaming variables v to $R(v)) \models$ always$(\lambda q. P(q \circ R))$.*

5.5 Specifying Ensemble Virtual Synchrony

To demonstrate the use of our new version of IO, we present the specification of Ensemble Virtual Synchrony (EVS). In the EVS model, processes join together to form *views*, which are sets of processes that vary over time as processes join and leave. Views are thought of informally as active multicast domains; when a process fails, or when the network becomes partitioned, a view may be split into several smaller views. When processes are created, or when the network heals, multiple views may merge into one. Each process has its own version of the view it is in. When a process replaces its view $view_1$ with a new view $view_2$, we say the process *installs* view $view_2$. Virtual synchrony provides the following informal properties on *views* and messages:

EVSsingle At any time a process belongs to exactly one view.

EVSself If a process installs a view, it belongs to the view.

EVSvieworder At any process, views are installed in increasing order of view identifier.

EVSnonoverlap If two processes install the same view, their previous views are either the same or they are disjoint.

EVSviewmessage All delivered messages are delivered in the view in which they were sent.

EVSfifo Messages between any two processes in a view are delivered in FIFO order.

EVSsync Any two process that install a view v_2, both with preceding view v_1, deliver the same messages in view v_1.

We can categorize these properties in three parts: message delivery (EVSfifo, EVSviewmessage), view properties (EVSsingle, EVSself, EVSvieworder, EVSnonoverlap), and message/view properties (EVSsync).

We can specify the message delivery properties using the FIFO automaton. Let $\mathit{VID} \subseteq \mathit{Label}$ be the type of views, and let $\mathit{PID} \subseteq \mathit{Label}$ be the type of processes. We implement the *Label* type as a recursive type including names and closed under pairing (which we used for subscripting by process and view), to get the following automaton:

$$\text{EVS\_FIFO} \quad = \quad \bigcap_{v \in \mathit{VID}} \bigcap_{p,q \in \mathit{PID}} . \; (\text{FIFO} \quad \begin{array}{l} \text{renaming} \\ \text{action SEND}(m) \text{ to EVS-SEND}_{p,v}(m) \\ \text{action RECV}(m) \text{ to EVS-RECV}_{q,p,v}(m) \\ \text{variable } \mathit{sent} \text{ to } \mathit{sent}_{p,v} \\ \text{variable } \mathit{received} \text{ to } \mathit{received}_{q,p,v}). \end{array} \qquad (3)$$

The EVSfifo property follows trivially from the FIFO automaton. The EVSviewmessage property follows because FIFO channels are only established within views (they do not cross views).

For the *view* properties, we introduce a new action EVS-NEWVIEW$_p(v)$, which delivers the view $v \in VID \times PID\ Set$ to process $p \in PID$. We refer to the view identifier as $v.id$ and the set of processes that belong to the view as $v.set$. We also need to introduce a history variable $\textit{all-viewids}_p$, for each process $p \in PID$, containing the set of views ever delivered to process p. Several derived variables are defined in terms of $\textit{all-views}_p$:

- $\textit{current-view}_p$ is the view in $\textit{all-views}_p$ with the largest identifier,
- $\textit{pred-view}_{p,v}$ is the view in $\textit{all-views}_p$ with identifier strictly smaller than $v.id$ if such a view exists, otherwise $\perp$, where $\perp \notin VID$ is a constant.

Given these definitions the view part of EVS is shown below.

EVS_VIEW

State: for each $p \in PID$:
  $\textit{all-viewids}_p \in View\ Set$, initially $\{v_p\}$

EVS-NEWVIEW$_p(v)$

Pre: let $v' = \textit{current-view}_p$ in
  $v.id > v'.id$ ①
  $p \in v.set$ ②
  $\forall q \in v.set$. ③
    if $\textit{pred-view}_{q,v} \neq \perp$ then
      $\textit{pred-view}_{q,v} = v' \vee \textit{pred-view}_{q,v}.set \cap v'.set = \{\}$

Eff: $\textit{all-views}_p = \textit{all-views}_p \cup \{v\}$

The precondition for EVS-NEWVIEW$(v)_p$ has several parts, one for each of the view properties. Part ① guarantees that views are delivered in ascending order (EVSvieworder), and the definition of $\textit{current-view}_p$ defines exactly one current view per process (EVSsingle). Part ② ensures that a process belongs to all its view (EVSself). Part ③ sets up the condition on previous views: if another process q has installed view v, then it either has the same previous view, or its previous view was disjoint from v (EVSnonoverlap).

The final part of EVS relates views and message ordering. For this automaton, we need to express the relation between the FIFO and VIEW automata, and we include state variables $\textit{received}_{q,p,v}$ and $\textit{all-views}_p$. The VIEW_MSG automaton is shown below.

VIEW_MSG

State: for $p, q \in PID$, $v \in View$:
  $\textit{received}_{p,q,v} \in \mathcal{M}\ List$
 for $p \in PID$:
  $\textit{all-views}_p \in View\ Set$

EVS-NEWVIEW$_p(v)$

Pre: let $v' = \textit{current-view}_p$ in
  $\forall q \in v.set$:
    $\forall r \in v'.set$. if $\textit{pred-view}_{q,v} = v'$ then
      $\textit{received}_{r,p,v'.id} = \textit{received}_{r,q,v'.id}$

The VIEW_MSG automaton introduces a new precondition for delivering a view: each process q with the same preceeding view v' receives exactly the same messages from each process $r \in v'.set$ (they EVSsync property).

These three automata specify the properties of EVS individually. The final step is combine them into the complete specification of EVS.

$$EVS \quad \equiv \quad \text{GROUP\_FIFO} \cap \text{VIEW} \cap \text{VIEW\_MSG}. \tag{4}$$

This construction identifies the $\textit{received}_{p,q,v}$ variables of the GROUP_FIFO and VIEW_MSG automata, and the $\textit{all-views}_p$ variables of the VIEW and VIEW_MSG automata. The properties of EVS are the conjunction of the properties of the three parts, forming a complete specification of EVS.

Abadi and Lamport [1] have explored composition in TLA using *assume-guarantee* specifications. We describe our systems with automata, rather than a temporal *logic* like TLA, because we can use a single language to represent both programs and specifications. However, our formalism shares the same problem space with TLA. In fact, our safety theorem $(A \models \mathit{always}\, P) \Rightarrow (A \cap B \models \mathit{always}\, P)$ is currently too weak: while the intersection $A \cap B$ has the safety properties of both A and B, additional safety properties may hold because of interference between the two automata. It is likely that we can use the assume-guarantee style of reasoning to extend our framework with a more complete result.

6 Conclusion

6.1 Objects

Class theory is only one aspect of the type system for object-oriented programming. It is possible to extend these ideas to account for other aspects of objects. Karl Crary has written "Simple, Efficient Object Encoding Using Intersection Types"[16], available at his web page, and the second author has written a draft article available at his web page, "A Predicative Type-Theoretic Interpretation of Objects." In addition Pavel Naumov has shown, in his PhD thesis, how to define Java's recursive types in Nuprl, *Formalizing Reference Types in Nuprl* [33].

Other aspects of Java's class theory can easily be expressed in our type theory. For example, we can include specific methods in a class by specifying that a field names a specific function. To define classes with specific methods we constrain elements of a type to a particular function, for example $\{x : T; x\_def : x = t\}$ for t a particular element of T. In this setting, method override is just a function from records to records.

For example, if the records defined by $Sig : Label \rightarrow Type$, then a map $F : (Label \rightarrow Type) \rightarrow (Label \rightarrow Type)$ can define an override. Let $\{Sig\}$ be the record $i : Label \rightarrow Sig(i)$, then $\{F(Sig)\}$ is the new class. For example, we could update the method t to t', perhaps writing this as: $\{x : T; x\_def : x = t\}\ [x := t]$.

Java has many restrictions dictated by the need to compile efficiently. For example, a class can have only one superclass, and abstract classes cannot be instantiated. Also the primative classes look like our types. It is pleasing to see how class theory can be developed without the restrictions of current compiler technology. This article has given a glimpse of such a theory.

6.2 Ludics

At the Marktoberdorf summer school of 1999, Jean-Yves Girard introduced a new logic called Ludics in which he took into account an analysis of space. It turns out that there is a strong connection to our class theory based on the following points of agreement. Girard also introduces an intersection type with logical meaning, he can define with it as a type like our Top in the same way, $\cap\ x : void.x$. He also uses a subtyping relation that agrees with ours. Finally, and most significantly, he introduces a naming mechanism, like our Labels, that allows naming objects. He says that objects with names are *located*. He uses localization to show how to define $A \times B$ from intersection. This turns out to be exactly these abbreviations:

$$\{x : A\} \ \cap\ \{y : B\} \ = \ \{x : A; y : B\} \quad \text{if} \quad x \neq y, \text{ and this is isomorphic to } A \times B$$

$$\{x : A\} \ \cap\ \{y : B\} \ = \ \{x : A \ \cap\ B\} \quad \text{if} \quad x = y, \text{ and this is isomorphic to } A \ \cap\ B.$$

Duality is important in Girard's theory, and intersection has a dual, namely union. We have defined a union type as well, $A \ \cup\ B$, but it did not play a role in this article. The type $A \ \cup\ B$ consists of the elements of A and the elements of B, and we say $a = b$ in $A \ \cup\ B$ when $a = b$ in A or when $a = b$ in B or when there is a c such that $a = c$ in A and $c = b$ in B.

A Appendix: Automata in NuPRL Type Theory

Mark Bickford, Odyssey Research Associates, Ithaca, NY, USA

A.1 Introduction

The aim of this appendix is to present some of the definitions and theorems of our NuPRL IO-automata theory as they appear in the NuPRL library. The IO-automata theory has two parts. The first part defines the semantic domain of automata, which we call state-machines, and operations on them and theorems about them. The second part defines a syntax for IO-automata and infinitary logic and a meaning function that maps the syntactic forms, which we call automata, to the semantic state-machines. In this appendix we present only some parts of the first part of the theory, the definitions of the state-machine type and operations on it.

We use a "bottom-up" style of presentation, giving the basic concepts first and progressing to the complex. The basics of NuPRL type theory, in particular, an understanding of dependent function, dependent product, intersection type, and set type, are assumed.

A.2 Patterns and Labels

We need a basic type of labels that includes strings and integers and is closed under pairing so that we can form labels like `sent[p]` and x_5. In order to define operations like matching and substitution on our labels we make a more general type of patterns that also includes pattern variables. The labels are then the subtype of "ground" patterns.

These two NuPRL abstractions

```
ptn_con(T) ==  Atom + ℤ + Atom + T × T
Pattern ==  rec(T.ptn_con(T))
```

make the type `Pattern` a recursive type that satisfies

```
Pattern ==  Atom + ℤ + Atom + Pattern × Pattern
```

The first two components of `Pattern` represent the atomic strings and integers that we want as labels. The third component represents the pattern variables, and the fourth component represents the label-pairs, which we write as `x[y]`.

Then we define the ground patterns as the ones that do not mention varaibles.

```
ground_ptn(p) == Case(p)
                  Case ptn_var(v) => ff
                  Case ptn_pr(<x, y>) =>
                    ground_ptn(x) ∧_b ground_ptn(y)
                  Default => tt
∀p:Pattern. ground_ptn(p) ∈ 𝔹
Label ==  {p:Pattern| ground_ptn(p)}
```

A.3 Declarations and Records

A declaration is an assignment of types to labels.

```
Decl ==  Label ⟶ 𝕌
```

The empty declaration assigns the type `Top` to every label; the base declaration `x:T` assigns `T` to `x` and `Top` to the rest; and we can combine declarations by using type intersection.

```
      ==  λx.Top
    x:T ==  λa.if a = x then T else Top fi
    d1;d2 ==  λx.d1 x ∩ d2 x
    (D[i] for i ∈ I) ==  λx.(∩i:I.D[i] x)
```

A renaming is a map `f` of type `Label → Label`. If declaration `d` assigns `T` to `x` and if `f x = z`, then the renamed declaration, `d o f`, should assign `T` to `z`. Rather than restrict renamings to be one to one we also handle the case of many to one renamings. If declaration `d` assigns `T1` to `x` and `T2` to `y`, and if `f x = f y = z`, then the renamed declaration, `d o f`, should assign `T1 ∩ T2` to `z`. So, we define the renaming operation on declarations as follows

```
    d o f ==  λx.(∩y:Label.if x = f y then d y else Top fi )
```

The record `{d}` defined by declaration `d` is the dependent function

```
    {d} ==  l:Label → d l
```

and record selection is just application

```
    r.l ==  r l
```

If `f` is a renaming and `r` is a record, then `r o f`, where the operation is the usual function composition, is another record. We have the following theorem about record types and renaming

```
    ∀d:Decl. ∀f:Label → Label.    ∀r:{d o f}.  r o f ∈ {d}
```

A.4 Sigma types

A dual to the record type is the "sigma" type. It is a dependent product defined by a declaration `d`. Members of this type are pairs of a kind and a value where the type of the value depends on the kind.

```
    (Σd) ==  l:Label × d l
    kind(a) ==  a.1
    value(a) ==  a.2
```

The renaming theorem for sigma types is

```
    ∀d:Decl. ∀f:Label → Label.    ∀a:(Σd o f). ∀l:Label.
        f l = kind(a) ⇒ <l, value(a)> ∈ (Σd)
```

A.5 State Machines

Our state machines are labeled transition systems. They are parameterized by two declarations, `da` and `ds`. Declaration `ds`, declares the types of the state varaibles, and the state of the machine is a member of the corresponding record type `{ds}`. Declaration `da` declares the types of the actions that label the transitions. Each action is a kind-value pair in the sigma type `(Σda)` corresponding to `da`. The value components of the actions represent input values.

```
SM ==  da:Decl × ds:Decl ×
       init:({ds} → ℙ) ×
       ({ds} → (Σda) → {ds}   → ℙ)
```

A state machine `M ∈SM` thus has four components, its two declaration parameters, an initial state predicate and a labeled transition relation. It also has its corresponding state and action types.

```
M.da ==  M.1
M.ds ==  M.2.1
M.init ==  M.2.2.1
M.trans ==  M.2.2.2
M.state ==  {M.ds}
M.action ==  (ΣM.da)
```

A.6 State Machine Composition

Using the combinators already defined for declarations, we can define the composition of two state machines and the composition of and indexed family of state machines.

```
(A ∩ B) ==
      <A.da;B.da, A.ds;B.ds,
       λs.A.init s ∧ B.init s,
       λs1,act,s2.A.trans s1 act s2 ∧ B.trans s1 act s2>
∩i:I. M[i] ==
      <M[i].da for i ∈ I, M[i].ds for i ∈ I,
       λs.∀i:I. M[i].init s,
       λs1,a,s2.∀i:I. M[i].trans s1 a s2>
```

A.7 State Machine Renaming

We define separate operations for renaming of state variables, which we write as `(M o f)`, and for renaming of actions, which we write as `(f o M)`

```
(M o f) ==
   <M.da, M.ds o f,
    λs.M.init (s o f),
    λs1,a,s2.M.trans (s1 o f) a (s2 o f)>

(f o M) ==
    <M.da o f, M.ds,
     M.init,
     λs1,a,s2.
     ∃l:Label. (kind(a) = f l) c∧
                 (M.trans s1 <l, value(a)> s2)>
```

A.8 Preservation of Invariants

We define the notion of invariant in the usual way

```
n_reachable(M;n;x) ==
  if (n =z 0) then M.init x
```

```
  else ∃x':M.state ∃a:M.action
       n_reachable(M;n - 1;x')
      ∧ M.trans x' a x

reachable(M;x) ==  ∃n:ℕ. n_reachable(M;n;x)
M |= always s.(I[s]) ==  ∀s:M.state. reachable(M;s)     ⇒ I[s]
```

Then we have the following preservation theorems for the composition and renaming operators.

```
∀A,B:SM. ∀Inv1:A.state → ℙ.    ∀Inv2:B.state → ℙ.
A |= always s.(Inv1[s])
⇒ B |= always s.(Inv2[s])
⇒ (A ∩ B) |= always s.(Inv1[s] ∧ Inv2[s])

∀I:𝕌. ∀M:I → SM. ∀Inv:i:I   → M[i].state → ℙ.
(∀i:I. M[i] |= always s.(Inv[i;s]))
⇒ (∩i:I.M[i]) |= always s.(∀i:I. Inv[i;s])

∀M:SM. ∀f:Label → Label.    ∀I:M.state → ℙ.
M |= always s.(I[s]) ⇒ (M o f) |= always s.(I[s o f])

∀M:SM. ∀f:Label → Label.    ∀I:M.state → ℙ.
M |= always s.(I[s]) ⇒ (f o M) |= always s.(I[s])
```

A.9 Refinement

To define a refinment relation between state machines, we need a more detailed definition of reachability, the reachable-from relation.

```
n_reachable_from(M;n;s;x) ==
 if (n =z 0) then x = s
 else ∃x':M.state. ∃a:M.action
   n_reachable_from(M;n - 1;s;x') ∧ M.trans x' a x

 s --M→ x ==  ∃n:ℕ.    n_reachable_from(M;n;s;x)
```

Then we define the refinement of `B` by `A` via relartion `R` as follows

```
A ≤ B via s,s'.R[s; s'] ==
(∀s:A.state. A.init s
  ⇒ (∃s':B.state. reachable(B;s') ∧ R[s; s']))
  ∧
(∀s1,s2:A.state. ∀a:A.action. ∀s1':B.state
  reachable(A;s1)
 ⇒ A.trans s1 a s2
 ⇒ R[s1; s1']
 ⇒ (∃s2':B.state. s1' --B→   s2' ∧ R[s2; s2']))
```

The preservation theorem for refinement is

```
∀A,B:SM. ∀R:A.state → B.state     → ℙ. ∀I:B.state → ℙ.
 B |= always s.(I[s])
⇒ A ≤ B via s,s'.R[s;s']
⇒ A |= always s.(∃s':B.state. R[s;s'] ∧ I[s'])
```

References

[1] Martin Abadi and Leslie Lamport. Conjoining specifications. *ACM Toplas*, 17(3), May 1995.

[2] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 215–224. IEEE, June 1987.

[3] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[4] Paul Bernays and A. A. Fraenkel. *Axiomatic Set Theory.* North-Holland Publishing Company, Amsterdam, 1958.

[5] Gustavo Betarte and Alvaro Tasistro. *Extension of Martin Löf's type theory with record types and subtyping* chapter 2, pages 21–39, in Twenty-Five Years of Constructive Type Theory. Oxford Science Publications, 1999.

[6] Mark Bickford and Jason Hickey. An object-oriented approach to verifying group communication systems. Department of Computer Science, Cornell University, 1998.

[7] N. Bourbaki. *Elements of Mathematics, Algebra, Volume 1.* Addison-Wesley, Reading, MA, 1968.

[8] Luca Cardelli. Extensible records in a pure calculus of subtyping. In Carl. A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics and Language Design.* MIT Press, 1994. A preliminary version appeared as SRC Research Report No. 81, 1992.

[9] Robert L. Constable. Experience using type theory as a foundation for computer science. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 266–279. LICS, June 1995.

[10] Robert. L. Constable. The structure of Nuprl's type theory. In Helmut Schwichtenberg, editor, *Logic of Computation*, volume 157 of *Series F: Computer and Systems Sciences*, pages 123–156, Berlin, 1997. NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 25–August 6, 1995, Springer.

[11] Robert L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter X, pages 683–786. Elsevier Science B.V., 1998.

[12] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[13] Robert L. Constable, Paul B. Jackson, Pavel Naumov, and Juan Uribe. Constructively formalizing automata. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, Cambridge, 1998.

[14] Robert L. Constable and D.R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[15] Karl Crary. Foundations for the implementation of higher-order subtyping. In *1997 ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, June 1997. to appear.

[16] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report TR98-1675, Department of Computer Science, Cornell University, April 1998.

[17] Karl Crary. *Type–Theoretic methodology for practical programming languages.* PhD thesis, Cornell University, August 1998.

[18] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958.

[19] J-Y. Girard. On the meaning of the logical rules I: syntax vs semantics. Institut de Math, Marseille, 1998.

[20] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. *Twenty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, January 1994.

[21] Robert Harper and John C. Mitchell. On the type structure of standard ml. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[22] Jason J. Hickey. Formal objects in type theory using very dependent types. Available through FOOL 3 web set on `www.cs.williams.edu`.

[23] C. A. R. Hoare. Notes on data structuring. In *Structured Programming.* Academic Press, New York, 1972.

[24] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[25] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, of Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, Berlin, 1996.

[26] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of Lecture Notes in Computer Science, pages 85–101. Springer-Verlag, Berlin, 1996.

[27] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert Constable. Building reliable, high-performance communication systems from components. *Proc. of the 17th ACM Symposium on Operating System Principles*, pages 80–92, December 1999.

[28] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[29] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, September 1989.

[30] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[31] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[32] Per Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes.* Bibliopolis, Napoli, 1984.

[33] P. Naumov. *Formalizing Reference Types in NuPRL.* PhD thesis, Cornell University, August 1998.

[34] Douglas R. Smith. Constructing specification morphisms. *Journal of symbolic Computation, Special Issue on Automatic Programming*, 16(5-6):571–606, 1993.

[35] Douglas R. Smith. Toward a classification approach to design. *Proceedings of the Fiftieth International Conference on Algebraic Methodology and Software Technology, AMAST'96, LNCS*, pages 62–84, 1996. Springer Verlag.

[36] Scott F. Smith and Robert L. Constable. Partial objects in constructive type theory. In *Proceedings of Second Symposium on Logic in Comp. Sci.*, pages 183–93. IEEE, Washington, D.C., 1987.