Formal Abstract Data Types

Jason J. Hickey

December 7, 1995

Abstract

Current constructive type theories are powerful systems for describing mathematical objects with complex dependencies between types and computational values, making them expressive enough to encompass large ares of mathematics and programming. However, as the body of formal knowledge in the type theory expands, the problem of managing mathematical domains and their proofs becomes increasingly significant. Though the objects of the theory are formal, the domains are not. In this paper, we show how to apply the methods of formal data abstraction to the organization of the mathematical domains. In the process we expand the tools of data abstraction to include reusability and namespace organization, providing an environment that can be used for defining objects within a domain, for organizing domains within a type theory, and for organizing theories within a system. This environment will require extending the expressivness of inductive definitions within the type theory to include the dependent characteristics of type theoretical domains.

1 Introduction

In a practical sense, modern constructive type theories are undergoing a crisis of scale. While it is possible to formalize significant areas of mathematics and programming, in most cases the effort involved is still much greater than is practical. Yet the application of formal methods to large practical problems continues to hold a great deal of appeal: as programs get larger, they become more difficult to reason about, making automated assistance even more desirable. These contradicting goals make it tempting to use type theories that favor practical techniques over semantical meaning, or that abandon formal methods entirely and rely on informal proof procedures.

Of course, informal methods will always be a cornerstone of automated verification and synthesis. Such techniques must be used when it come to dealing with facts that are "obvious" in some way. However, the verification problem eventually repeats itself; a verification procedure that does not catch errors is not very useful, and a procedure that signals false errors is often worse. If we could prove that the verification procedure is correct, the payoff would be big.

The problem of scale in type theories has two aspects. First, formal proofs are often much larger and more difficult than proofs on paper, because the formal system has no idea of what is "obvious." Second, the type theories are intrinsically not scalable; even though the objects of the type theory are formal, their organization is *ad-hoc*. We don't address the first issue in this paper. It is a difficult problem, and it may only be alleviated by finding more powerful algorithms for proof search. The second issue is more tractable. There is a great deal of experience, both formal and informal, on using organizational tools to deal with the problems of scale. From abstract algebra, we have formal definitions for modularity and abstract data types. From the classification methods of taxonomy, we get an orthogonal organization scheme that we can use to provide abstract data types with inheritance and re-use (making them "object-oriented"). Finally, from software engineering, we can use experience in naming to classify and identify the objects we create.

Before we proceed, it is useful to provide some terminology. We use the word "object" to refer to data that belongs to a "signature," consisting of a collection of named "methods." The methods each have an arbitrary type, and the signature is the product of those types. Signatures are themselves types, and may be used to specify the method types of further objects. We differ from the standard (for example Abadi and Cardelli [2]) in that the methods are simple types, not dependent upon the signature being defined. By using this loose definition, we will be able to apply our treatment to each part of the type theory, including the type theory itself and the *theories* (or *mathematical domains*) that are defined within it. We also make this restriction so that we can provide a well-defined semantic meaning to the type of a signature that will not rely on a fixed point construction.

The type that we will assign to the signature of an object will be a dependent record type. That is, a signature will consist of a collection of labeled types that depend on one another in some well-founded way. As it happens, this will require that we extend the type theory with a suitable "very-dependent" type, which is an interesting type-theoretical problem in itself. In the following sections of the paper, we begin with an example that illustrates the features we will need from the type theory. Following that, we describe a very-dependent type extension to the NuPRL type theory, and we cover parts of the type theory to give some intuition about the extension. In section 3, we develop the objects and give an informal account of an "object system." In section 4, we extend the object system to include specifications of mathematical domains. Following that, we cover some related work, and we present the main proposal of this thesis. The Appendix covers the NuPRL type theory in some detail, although the detail is not necessary for the work in this paper.

1.1 Module example

Before proposing an extension to the type theory, it will be useful to look at an example of a module specification that we would like to describe. This will point out some of the features that we will want from the type theory.

An obvious example comes from abstract algebra (we would hope that algebraic objects have easy descriptions). One of the first objects studied in an elementary algebra course is the semi-group, or monoid. The description is roughly as follows: a monoid is a triple $(S, \oplus, e)$, where S is a set called the *carrier*, $\oplus: S \times S \to S$ is a binary operation on the set, and $e \in S$ is an identity of $\oplus$. Since we will be giving a formal account in a constructive type theory, the carrier will be specified by a type *car* and a decidable equality $=$ (the type denoting the carrier will, in general, have a coarser equality than the set we are interested in). Within the type theory, the monoid can be specified with the following type:

$$
\boxed{
\begin{array}{lcl}
\text{Monoid} & == & car\colon Type \\
 & \times & =\colon car \to car \to Prop \\
 & \times & eq\text{–}prop_{ref}\colon \forall x\colon car.x = x \\
 & \times & eq\text{–}prop_{sym}\colon \forall x, y\colon car.x = y \Rightarrow y = x \\
 & \times & eq\text{–}prop_{trans}\colon \forall x, y, z\colon car.x = y \Rightarrow y = z \Rightarrow x = z \\
 & \times & \oplus\colon car \to car \to car \\
 & \times & assoc\text{–}prop\colon \forall x, y, z\colon car.(x \oplus y) \oplus z = x \oplus (y \oplus z) \\
 & \times & e\colon car \\
 & \times & identity\text{–}prop\colon \forall x\colon car.x \oplus e = x
\end{array}
}
$$

The dependencies can be determined syntactically: every method depends on *car*, except *car* itself. The eq–prop$_*$ methods depend on $=$ but not on each other, etc. This particular signature can be expressed as 9–ary dependent cartesian product. The signature itself is just a type. If we wish to specify a group, we can just extend the signature of the monoid by adding the axiom that a group must have an inverse:

$$
\boxed{
\begin{array}{lcl}
\text{Group} & \textbf{is} & \text{Semi-group} \\
 & \times & inverse\text{–}prop\colon \forall x\colon car.\exists y\colon car.x \oplus y = 1
\end{array}
}
$$

In this case, the signature of the group is a binary product, containing the signature for the monoid, plus the *identity* axiom.

There are at least three problems that are immediately apparent:

1. There is no consistent signature type. Signatures are just types, which may be binary products, 9-ary products, or any other type. This affords considerable flexibility but little structure.

2. Coercions must be explicit. We would like to say that a group *is* a monoid, without having to provide the explicit coercion function. Put another way, we would like groups to be a subtype of monoid. And yet another way: we would like all theorems about groups to apply to monoids transparently.

3. There is no support for multiple inheritance (in fact there is no support for inheritance). The methods of the monoid are not directly available in the group. Furthermore, if we describe a module with two parents, the common ancestors won't be joined.

We can solve these problems by introducing a level of indirection. That is, we introduce a type of *descriptions* of signatures. This could simply be a list of labels and types assigned to the labels. We would like to have a type of all signature descriptions, so this would be a recursive type. Furthermore, the types associated with each label depend on inhabitants of previous types in the list. Thus, the ideal type for signature descriptions would be a dependent record. For instance, the signature description for the monoid would be the record:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{Monoid–description} == [} \\
\quad car, & Type; \\
\quad =, & car \to car \to Prop; \\
\quad eq\text{–}prop_{ref}, & \forall x{:}\, car. x = x; \\
\quad eq\text{–}prop_{sym}, & \forall x, y{:}\, car. x = y \Rightarrow y = x; \\
\quad eq\text{–}prop_{trans}, & \forall x, y, z{:}\, car. x = y \Rightarrow y = z \Rightarrow x = z; \\
\quad \oplus, & car \to car \to car; \\
\quad assoc, & \forall x, y, z{:}\, car. (x \oplus y) \oplus z = x \oplus (y \oplus z); \\
\quad e, & car; \\
\quad identity, & \forall x{:}\, car. x \oplus e = x \\
]
\end{array}
$$

The signature of the monoid is a dependent record type corresponding to this description, and monoids are dependent records inhabiting the signature. By constraining signatures in this way, we can define a generic set of operations that are well-formed on all objects having a signature, solving the first problem. The dependent records use transparent record subtyping. So, for instance, a group is a monoid, because it contains all the fields of a monoid. The coercion is implicit, because it just requires "forgetting" the extra field of the group, solving the second problem. We can solve the last problem by designing the signature generation algorithm so that it merges common ancestors. Using the dependent record type, method selection corresponds to field selection. Since the group is a monoid, the methods of the monoid are inherited by the group. The problem of multiple inheritance is not completely solved: if we consider rings (which contain a monoid and a group with a common carrier), then a ring can be coerced to a monoid in two distinct ways. We will discuss this issue in section .

Since the type theory doesn't provide a constructor for dependent records, that will be the first extension we will make. In the next section we generalize the goals of dependent records to a class of type we call "very-dependent" types. These types have the common feature of potentially describing an infinite number of dependencies.

2 Very-dependent type extension

Most constructive type theories have types that can express complex dependencies between types and values, or between types and types. These dependencies are crucial for providing precise specifications for computational objects. For example, the base type theory generally provides a dependent function type, often written as $x{:}\, A \to B$ or $[x{:}\, A]B$, where A is a type, and if x is an object inhabiting A, then B is a type. The type A describes the domain of the function, and $B(x)$ describes the range of the function on argument x. The dependent type can be used to provide a more precise specification for a set of functions than the undependent function type. For instance, using the dependent type we can define a class of natural number functions whose output is always larger than their input with the type $F \equiv i{:}\, \mathbb{N} \to \{(i+1)\ldots\}$. While each function described by F is also included in $\mathbb{N} \to \mathbb{N}$, clearly F is a more precise specification for the set of functions we are interested in.

Inductive types are used to describe recursive objects such as lists and trees. In NuPRL [7] simple inductive types are described with an inductive type constructor $\mu\,(X.T)$ where X is bound in the term T and represents the fixed point, i.e. $\mu\,(X.T) = T\,[\mu\,(X.T)\,/X]$. For this type to be meaningful according to its semantics, the term T must be a type that is monotonic in X for *any* type X. In

label	type
x_0	f_0
x_1	$f_1(x_0)$
x_2	$f_2(x_0)(x_1)$
x_3	$f_3(x_0)(x_1)(x_2)$
$\vdots$	$\vdots$

Figure 1: Telescope

Coq [18], a similar construction with a *syntactic* constraint can be used to define inductive types for the decidable type theory. Similarly, HOL [8] contains a mechanism for introducing primitive recursive types.

The inductive types are useful for a variety of purposes. They are used to define inductively defined predicates, relations, and also to define abstract data types for some degree of modularity. Most mathematical reasoning requires use of induction, and inductively defined types are fundamental to most modern computation systems. Driven by the essential importance of inductive types, some proof systems, notably it Coq, code most type constructors on top of the inductive type, so that the only type constructions needed are the inductive type and function space.

All of the forms of inductive types are used to describe recursive objects *without* type dependencies over the recursion. For example, in NuPRL, given a type T, we can describe a list of objects of type T with the type $\mu\,(X.\mathit{Unit} \mid T \times X)$, where *Unit* is a type inhabited by a single object. The objects in the list each have type T, and they are each independently described. If T were the set of integers, the list would contain integers having no relation to one another.

Now suppose we wish to describe recursive objects where there are dependencies over the recursion: for instance, lists of sorted integers. The more general form of the inductive type constructor in NuPRL allows a parameter to be passed to the construction, along with an initial value for the parameter. Parameterized types are defined with the construction $\mu\,(X, i.T; v)$, where X and i are bound in T, X represents the parameterized type fixed point, i represents the parameter value, T is the type construction, and v is the initial value of the parameter. Using the parameterized recursive type, we can describe increasing lists of natural numbers with the type

$$L_I \equiv \mu\,(i, X.\mathit{Unit} \mid j{:}\,\{i\ldots\} \times X\,(j+1)\,;0)\,.$$

Note that the dependencies are from the front of the list: the first element must be 0 or larger, the second must be larger than the first, and so on. We can define a constructor for this list as *cons*, where

$$\mathit{cons}(h,t) \equiv \mathit{inr}\,\langle h,t\rangle$$

This constructor creates objects of type L_I if, and only if, t is an list of increasing integers and it is empty, or its first element is larger than h. Thus, using this construction the second argument to *cons* must belong to a subtype of L_I that depends on the first argument.

Although the typing rule for *cons* is somewhat awkward, it is sufficiently practical that is can be used for normal program development. However, consider the type of *telescopes* as described by DeBruijn [?]. For our purposes, a telescope is a vector of objects as shown in Figure 1. The type of element x_i in the vector is given by a function f_i that produces a type given the first $i-1$ elements of the vector. That is, the first element x_0 of the vector has type f_0, the second element x_1 has type $f_1(x_0)$, the third $f_2(x_0)(x_1)$, and so on. If we are to give a type for the telescope we have two problems: what is the type of the type-function vector $\mathbf{f} = f_0, f_1, \ldots, f_n$, and what is the type of the telescope $\mathbf{x} = x_0, x_1, \ldots, x_n$?

Assuming that we can give a type for the type-function vector, and we have an actual type-function vector $\mathbf{f}$, the type of the telescope can be described with the parameterized recursive type

$$T \equiv \mu\,(i, X.\mathit{Unit} \mid x{:}\,(\mathit{nth}(\mathbf{f}, i.1)(i.2)) \times X((i.1)+1, \langle i.2, x\rangle); \langle 0, \cdot\rangle)$$

Here, the function $[\mathit{nth}{:}\star\ \mathit{list} \times \mathbb{N} \to \star]$ computes the n^{th} element of a list, and i has the informal type $j{:}\,\mathbb{N} \times f_0 \times f_1 \times \cdots \times f_j$.

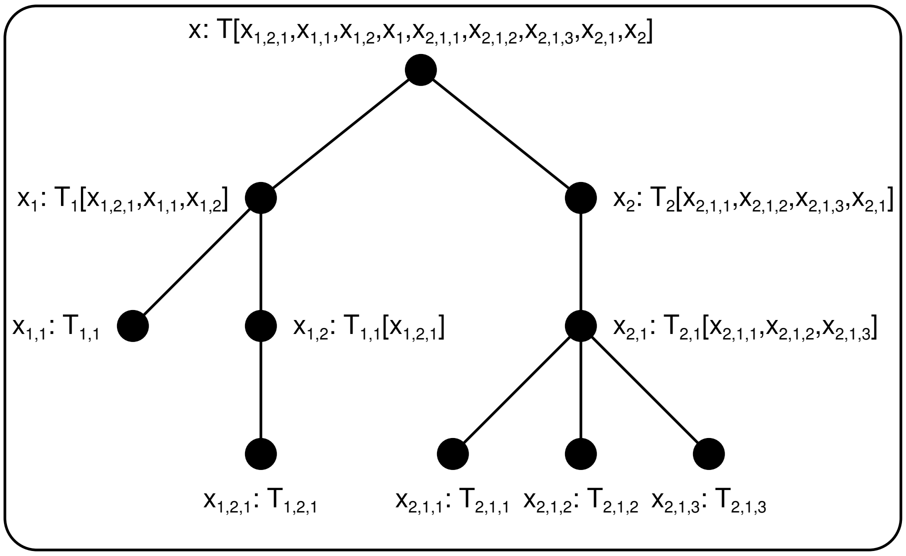

Figure 2: Tree of dependent types

There is no *cons* operation for a telescope, because it doesn't make sense to add an element to the telescope before x_0. Nor is it possible to give a type for telescopes represented in reverse order (at least using the parameterized recursive type). The dependencies between types and values in the telescope must proceed from the first element to the last, and telescopes must be constructed using an *append* operation.

The "direction" of the dependencies ultimately limits the expressivity of the parameterized recursive type. Suppose we wish to generalize the linear telescope to trees of values, where the type of the value of each vertex depends on the values of all the children, as shown in Figure 2. Using the parameterized recursive type it would be difficult to describe the tree because the description would necessarily begin at the leaves. Any such description would be so unnatural as to be practically unusable. The natural solution is to provide a type constructor that would allow the tree to be described from the root. Assuming these trees have type T, the root has children of type T, and it also has a type that depends on the specific children. We use this as the definition of a *very-dependent* type: that an object of the very-dependent type T contains subobjects that have type *dependent* upon proper subobjects of type T.

At first glance, it seems that we can define very-dependent types using the obvious dependent construction. For instance, to describe the list of increasing natural numbers, we could give the type:

$$L'_I \equiv \mu\,(X.\mathit{Unit} \mid l{:}\,X \times (\mathit{case}(l)\ \mathit{of}\ \mathit{inl}\,\_ \Rightarrow \mathbb{N} \mid \mathit{inr}\,\langle i, \_\rangle \Rightarrow \{\ldots i\}))$$

This type specifies that the *last* element of a list must be a natural number, and all other elements must be *smaller* than the next successive element. The *cons* constructor is the same as for the previous integer list, but now $cons(h, t)$ is well-defined if t inhabits L'_I and h is a natural number that is smaller then than the first element of t if t is not empty.

The problem with this construction is that the semantics of the μ recursive type require that the term $\mathit{Unit} \mid l{:}\,X \times (\mathit{case}(l)\ \mathit{of}\ \mathit{inl}\,\_ \Rightarrow \mathbb{N} \mid\Rightarrow \mid \mathit{inr}\,\langle i, \_\rangle\{\ldots i\})$ denote a type for *any* type X. But this clearly doesn't hold since the *case* expression is only defined if l belongs to a disjoint union.

It is critical here to understand that the *semantics* of the μ inductive type forbids very-dependent type constructions. The reason is that the fixed point is not bounded, except over *Type* (the collection of all types). In order for the fixed point to be well-defined, the definition must be well–formed for any value of the type variable X. This forbids dependent constructions because they require some structure on the dependen type. If we are to propose a solution, we will need to provide a different construction with its own semantics. Indeed, as we will show in section 2.2, we can give a different semantics for the μ construction that allows the type construction above.

In the next section, we explore the first extension to the type theory, the type of *very-dependent* functions. Although we don't normally think of function types as being recursive, we can see the correspondence by considering functions with domain $\mathbb{N}$. These functions are isomorphic to infinite lists, and the dependencies in the type of the elements of the list corresponds to having a range type for a value depend on "previous" values of the function. This concept can be extended to general well-founded domains.

2.1 Very-dependent function type

The normal dependent function type allows dependencies to be expressed between the argument of the function, and its range type. For instance, the NuPRL type $i{:}\mathbb{Z} \to \{i\ldots\}$ is the type of functions taking integer arguments and returning integer values that are at least as large as their argument. However, there are no dependencies between the types of values of the function on different arguments.

In order to introduce such dependencies, we introduce a new type constructor $\{f \mid x{:}A \to B[f,x]\}$, where f and x are bound in B. This corresponds to a function f with domain type A, and range $B[f,x]$ on argument $x \in A$.

2.1.1 Example of a very-dependent function

Before proceeding with a more formal treatment, it may be useful to present an example. Suppose we wish to encode an n-ary dependent product $x_1{:}T_1 \times x_2{:}T_2[x_1] \times \cdots \times x_n{:}T_n[x_1,\ldots,x_{n-1}]$. We can encode this type using the very-dependent function type, where the inhabitants are functions with domain $\{0\ldots(n-1)\}$, using the type

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

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

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

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

$$\mathit{napply}\,(f)\,(i)\,(G)\,(j) \equiv G\big(f(j)\big)\big(f(j+1)\big)\cdots\big(f(i-1)\big)$$

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

$$T_f(n) \equiv \{f \mid i{:}\{0\ldots(n-1)\} \to \mathit{napply}\,(f)\,(i)\,(F\,(i))\,(0)\}$$

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:

$$\begin{aligned} \mathit{nfun} \equiv \lambda F.\lambda i.\,\mathit{fix}(g.\lambda h.\lambda j.\ & \mathbf{if}\ j = i\ \mathbf{then} \\ & \qquad \mathbb{U}_i \\ & \mathbf{else} \\ & \qquad x{:}\,\mathit{napply}\,(h)\,(j)\,(F\,(j))\,(0) \to g(\lambda k.\,\mathbf{if}\ k = j\ \mathbf{then}\ x\ \mathbf{else}\ h\,(k))(j+1)\ ) \end{aligned}$$

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

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

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

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

The type for F is then:

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

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

$$\begin{array}{rcl} S_f & \equiv & n{:}\mathbb{N} \times T_f(n) \\ S_F & \equiv & n{:}\mathbb{N} \times T_F(n) \end{array} \tag{1}$$

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. From here we will cover a more formal treatment of the function type, beginning with its semantics.

2.1.2 Semantics of the very-dependent function type

The central issue in the semantics for the very-dependent function type $\{f \mid x{:}\, A \rightarrow B[f, x]\,\}$ is deciding what it means to use the function in its range. Is it possible to define the type of a function an an argument in terms of the function's value on the argument? The immediate inclination is to answer no—consider the term $\{f \mid i{:}\mathbb{Z} \rightarrow f\,(i)\}$. If this were a type, it would violate predicativity because, for instance, $f\,(0) \in f\,(0)$. Would this type, however, make sense in an impredicative system with some *Type* where *Type* $\in$ *Type*?

Even if we rule out impredicative types, we might want to allow some cyclic definitions. For instance, suppose we have some term that tests for computational convergence. Let $\{p\}\!\downarrow$ denote the fact that the program p converges. Consider the type T where,

$$T \equiv \{f \mid i{:}\mathbb{Z} \rightarrow \textbf{if}\ \{f\,(i)\}\!\downarrow \textbf{then}\ \mathbb{Z}\ \textbf{else}\ \mathit{Void}\,\}$$

This type might denote the partial functions with domain and range $\mathbb{Z}$.

In the next section, we will discuss 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 $\{f \mid x{:}A \to B\}$ 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 $\{f \mid x{:}A \to B\}$as follows.

Semantics 1 *The term $\{f \mid x{:}A \to B\}$ 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) *$\{f \mid x{:}\{a''{:}A \mid a'' < a'\} \to B\}$ 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 $\{f \mid x{:}\{a''{:}A \mid a'' < a'\} \to B\}$ reduces to $\{f \mid x{:}\,\mathit{Void} \to B\}$ which is a type for any term B. Similarly, the semantics requires that at the base case require that $B[g,a']$ be a type for the function g with domain *Void*, which means that B canot evaluate g on any argument. The definitions for γ and β are constructed mutually recursively for the entire domain A. We can summarize these points in the following theorem.

Theorem 1 *Semantics 1 are well-defined.*

Proof: The semantic definition is well-founded, because the membership constructions are defined by induction, based on the well-ordering of A.

We must show that the membership function is well-defined. Suppose we have a type $\{f \mid x{:}A \to B\}$ that is defined according to Semantics 1, and consider an element f where $\varphi(f)$. By item 3a, we know that $\gamma_a(f)$ and $\beta_f(f(a))$ for any $a \in A$. It follows that f is a function. Any element g, where $g = f$, must also be a function with domain A. By extensionality, $g(a) = f(a)$ for any element $a \in A$, and so $\beta_g(g(a))$ and $\gamma_a(g)$ hold. It follows that $g = f \in \{f \mid x{:}A \to B\}$. □

We must now determine rules for the type that reflect the semantics. The most difficult rule is for type membership. For a type $\{f \mid x{:}\{a''{:}A \mid a'' < a'\} \to B\}$ to be well-formed, the requirements are:

1. A must be a type,
2. A must be well-ordered according to some relation $<$,
3. B must be a type defined by induction on the well-ordered prefixes of A.

The fact that A is well-ordered is necessary to fit the semantic characterization, but the computational content of the proof of well-ordering is irrelevant. In the rule, we can discard the computational content of the proof. The type A is well-ordered iff it has an induction principle. We state this as a proposition:

$$\boxed{\mathit{WellFounded}_i\,(A;<) \quad\equiv\quad \forall P{:}A \to \mathbb{U}_i.\forall a_1{:}A.\big(\forall a_2{:}A.a_2 < a_1 \Rightarrow P\,(a_2)\big) \Rightarrow P\,(a_1) \Rightarrow \forall a{:}A.P\,(a)}$$

The proof of well-foundedness is then stated with the "set" type, in order to discard the proof:

$$\downarrow\{\mathit{WellFounded}_i\,(A;<)\} \equiv \{\mathit{Unit} \mid \mathit{WellFounded}_i\,(A;<)\}$$

The inhabitant "$\cdot$" exists only if $\mathit{WellFounded}_i\,(A;<)$ is true, but the proof of $\mathit{WellFounded}_i\,(A;<)$ is not actually saved.

The rules can be developed along these lines to give the rules of Table 1. The equality rule enforces the requirements of the previous paragraph. There is no formation rule, because the recursion in the

$H \vdash \{f_1 \mid x_1\colon A_1 \to B_1\} = \{f_2 \mid x_2\colon A_2 \to B_2\} \in \mathbb{U}_i$ ext $\cdot$
BY rfunctionEquality $R\ g\ y\ z$
$H \vdash A_1 = A_2 \in \mathbb{U}_i$
$H \vdash R = R \in A_1 \to A_1 \to \mathbb{U}_i$
$H \vdash \downarrow \{WellFounded_i\,(A_1; R)\}$
$H, y\colon A_1, g\colon \{f_1 \mid x_1\colon \{z\colon A_1 \mid R(z)(y)\} \to B_1\} \vdash B_1\,[g, y/f_1, x_1] = B_2\,[g, y/f_2, x_2] \in \mathbb{U}_i$

Equality rule

$H \vdash \lambda x_1.b_1 = \lambda x_2.b_2 \in \{f \mid x\colon A \to B\}$ ext $\cdot$
BY rfunctionLambdaEquality *level–exp* $\{i\}\ b$
$H, b\colon A \vdash b_1\,[b/x_1] = b_2\,[b/x_2] \in B\,[\lambda x_1.b, b/f, x]$
$H \vdash \{f \mid x\colon A \to B\} = \{f \mid x\colon A \to B\} \in \mathbb{U}_i$

Member equality (intensional)

$H \vdash f_1 = f_2 \in \{g \mid x\colon A \to B\}$ ext t
BY rfunctionExtensionality *level–exp* $\{i\}\ \big(\{g_1 \mid x_1\colon A_1 \to B_1\}\big)\ \big(\{g_2 \mid x_2\colon A_2 \to B_2\}\big)\ y$
$H \vdash \{g \mid x\colon A \to B\} = \{g \mid x\colon A \to B\}\,\mathbb{U}_i \in$
$H, y\colon A \vdash f_1\,(y) = f_2\,(y) \in B\,[f_1, x/g, x]$ ext t
$H \vdash f_1 = f_1 \in \{g_1 \mid x_1\colon A_1 \to B_1\}$
$H \vdash f_2 = f_2 \in \{g_2 \mid x_2\colon A_2 \to B_2\}$

Member equality (extensional)

$H \vdash \{f \mid x\colon A \to B\}$ ext $\lambda y.\ Y\,(\lambda g.b)$
BY rfunctionLambdaFormation *level–exp* $\{i\}\ R\ g\ y\ z$
$H \vdash A = A \in \mathbb{U}_i$
$H \vdash R = R \in A \to A \to \mathbb{U}_i$
$H \vdash \downarrow \{WellFounded_i\,(A; R)\}$
$H, y\colon A, g\colon \{f \mid x\colon \{z\colon A \mid R(z)(y)\} \to B\} \vdash B\,[g, y/f, x]$ ext b

Member formation

$H, f\colon \{g \mid x\colon A \to B\}, J \vdash T$ ext $t\,[f\,(a), \cdot/y, v]$
BY rfunctionElimination *assumption–index* $\{i\}\ a\ y\ v$
$H, f\colon \{g \mid x\colon A \to B\}, J \vdash a = a \in A$
$H, f\colon \{g \mid x\colon A \to B\}, J, y\colon B\,[f, a/g, x], v\colon y = f\,(a) \in B\,[f, a/g, x] \vdash T$ ext t

Elimination

$H \vdash f_1\,(a_1) = f_2\,(a_2) \in B\,[f_1, a_1/f, x]$ ext $\cdot$
BY rfunctionApplyEquality $\{f \mid x\colon A \to B\}$
$H \vdash f_1 = f_+2 \in \{f \mid x\colon A \to B\}$
$H \vdash a_1 = a_2 \in A$

Combinator equality

Table 1: Rules for very-dependent function with simple semantics

type definition requires that the entire type be constructed at once. For the extensional equality, the two terms being compared must be functions, they must return equal values on equal arguments, and the term of the equality must be a very dependent function type. For member equality rules, it is necessary to prove immediately that the the term of the equality is a type. This is contrary to the rules for dependent function equality. For example, in the rule for *lambdaEquality* in Table 9 it is possible to infer that the term $x\colon A \to B$ is a type from the semantics of the sequents. However, if we were to attempt a similar argument for *rfunctionLambdaEquality*, we would would wind up with the rule in Table 2. In the process of showing that B obeys the well-ordering on A, we would also be showing that B is a type. All of the subgoals necessary for proving the well-formedness of the equality type are present, so the shorter rule of Table 1 is adopted.

The rule for *rfunctionLambdaFormation* is interesting because its proof is by induction, and so the extract requires an induction combinator (The Y-combinator is used in the Table). The

$H \vdash \lambda x_1.b_1 = \lambda x_2.b_2 \in \{ f \mid x{:}\, A \to B \}$ ext $\cdot$
BY rfunctionLambdaEquality2 *level–exp* $\{i\}$ b
$H \vdash A = A \in \mathbb{U}_i$
$H \vdash R = R \in A \to A \to \mathbb{U}_i$
$H \vdash \downarrow \{ \mathit{WellFounded}_i\,(A; R)\}$
$H \vdash y{:}\, A_1, g{:}\, \{ f_1 \mid x_1{:}\, \{z{:}\, A_1 \mid R(z)(y) \} \to B_1 \}\, B_1\,[g, y/f_1, x_1] = B_2\,[g, y/f_2, x_2] \in \mathbb{U}_i$
$H, b{:}\, A \vdash b_1\,[b/x_1] = b_2\,[b/x_2] \in B\,[\lambda x_1.b, b/f, x]$

Member equality (intensional)

Table 2: Attempted rule for *rfunctionLambdaEquality*

termination is enforced by the well-foundedness of the domain. Interpreted as a proposition, the type $\{ f \mid x{:}\, A \to B \}$ is a recursive universal proposition. That is, if it is true, then for all $x \in A$, the proposition B holds *assuming* that the universal implication is true on smaller values of the argument. In essence, this *is* an argument by induction, and the recursive proof extract makes sense.

Note that it is not necessary to include the well-order of the domain in the very-dependent function type. That well-ordering is property that is used for the well-formedness argument where it is used to show that the definition is not cyclic; it is useless anywhere else. Also, the specific well-order is not important. There may be more than one well-order for which the type is well-formed. If so, there is more than one termination argument, but the termination holds in any case.

As we stated before, the semantics of this section are quite simple—and quite conservative. It is not necessary that the domain be well-ordered everywhere—just on those parts that are used to define the range.

2.2 Very dependent recursive types

Another avenue we can explore is defining general very-dependent recursive types. By very-dependent recursive types, we mean the normal parameterized recursive type $\mu\,(i, X.T[i, X]; v)$, where the type construction T is allowed to depend on X belonging to the recursive type definition. In the characterizations that we explore, we do no enforce a monotonicity condition on the type construction T, with the consequence that out characterizations are not based on a fixed–point.

The first characterization is one that requires no extension to the type theory. Instead of defining a new primitive type constructor, we assign an index of unrolling to each element of the type, and we use the type

$$i{:}\, \mathbb{N} \times \mathit{unroll}\,(x, X.T[i, X]; v)^i$$

where we define

$$\begin{array}{rcl} \mathit{unroll}\,(x, X.T[i, X]; v)^0 & \equiv & \mathit{Void} \\ \mathit{unroll}\,(x, X.T[i, X]; v)^i & \equiv & T[v, \lambda v.\mathit{unroll}\,(x, X.T[i, X]; v)]^{i-1} \quad (\text{for } i > 0) \end{array}$$

Essentially, the inhabitants are required to carry along the index of the unrolling that they belong to, and if the indexing is to be relatively transparent, the constructors must modify the indexes. For instance, the type of lists of monotonically increasing sequences of integers would be the type:

$$\begin{array}{l} \mu\,(T.\,\mathit{Unit} \mid l{:}\, T \times \mathit{case}\; l\; \mathit{of}\; \mathit{inl} \cdot \Rightarrow \mathbb{Z} \mid \langle \_, i \rangle \Rightarrow \{i + 1 \ldots\}) \\ \quad \equiv \quad i{:}\, \mathbb{N} \times \mathit{unroll}\,(T.\,\mathit{Unit} \mid l{:}\, T \times \mathit{case}\; l\; \mathit{of}\; \mathit{inl} \cdot \Rightarrow \mathbb{Z} \mid \langle \_, j \rangle \Rightarrow \{j + 1 \ldots\})^i \end{array}$$

The [] element is *inl* $\cdot$ and the *cons* operation is

$$h :: t \quad = \quad \mathit{case}\; t\; \mathit{of}\; \langle i, l \rangle \Rightarrow \langle i + 1, \mathit{inr}\,\langle l, h \rangle \rangle$$

Induction of lists becomes induction on their lengths.

While this coding is sufficient for most constructions, it is intrinsically unsatisfying. The indices of unrolling are used only so that we can give the construction a type. In fact, there is enough information in the construction that we could compute the index if necessary. The requirement of indices confilct with a philosophy we normally adopt: programs should not be required to compute type information. The reason for this philosophy is the thought that the typing of an algorithm should not interfere with its efficiency; the computation of type information is inherently useless.

There are other pssiblities. For instance, we can equate the meaning of a recursive type with its infinite unrolling, or with the uniuon of all its unrollings. These are areas under development, and are not discussed in this paper.

3 Formal Abstract Data Types

In the previous section we described an extension to the type theory to allow very–dependent types. We will use these extensions to specify abstract data types that allow transparent subtyping, with inheritance. All existing systems that are termed "object-oriented" have features for modularity and re-use, and so our development lies along the line of object-oriented systems. However, we plan to take the methodology a little further and apply it to the formal system itself.

The work in the section is experimental and not fully developed. We sketch only the basic outline of a fully modular system, but the details are left for further research. In fact, this research is quite exciting, with the goals of finding a formal system that we can *reason* about at a high level. However, it should not be expected that an "object-oriented" methodology is a *panacea* for the problems that software systems are facing today. The human—computer interaction is bogged down by an *excess* of formality. The programs we write today are extremely formal, requiring careful attention to details. The goal of object–orientation is to raise the level of formalism so that it deals with high-level user-defined (thus, hopefully, intuitive) modules. The inheretance aspect places value re-use, so that programmers are required to code only those parts of the program that differ from a *standard* coding. However, the program is still a formal object, performing exactly the tasks the programmer specified, whether expected or not.

In any case, the object-oriented mothodology is directed at the problem of *scale*, which is the main problem that we are addressing. In this section, we describe (based on the result of the previous sections) how objects in the type system can be organized into formal objects that allow for re-use. After that, we show that an "object" is the same as a "theory" (or mathematical domain), according to an interpretation based on the Curry–Howard isomorphism.

3.1 Formal components of an "object–system"

By the term "object–system," we mean any system that provides the following components:

1. Abstract data types, specified by signatures, consisting of "methods" that implement and verify components of the abstract data type.

2. Subtyping, with automatic inheritance of the methods of an abstract data type.

3. Optionally, method override, where methods may create copies of an object with new values for some of the methods.

There are other features that are common to many object–oriented implementations that we do not attempt to explain. For instance, many typed object-systems also provide operator overloading (also called "ad-hoc polymorphism"). In some systems method invocation is treated as message passing. While these features mesh well with an object–oriented style, they do not occur in all implementations, and they don't seem to be essential, as opposed to the modularity and inheritance features, which are common to all object systems. While we do not provide these extra features, our treatment will not preclude them.

Method override is one of the more controversial components of the object system, because of its semi–imperative nature. Method override is useful in a system where the methods of an object are allowed to refer to one another in their implementations. When a method is overridden, the other methods of the object are updated automatically to refer to the new value. Despite its imperitive nature, the functional coding is quite reasonable. In the first–order object system of this section we will include method override, and method interdependence, as practical features of the object system.

Method interdependence is really a practical feature of the object system, since any implementation can be unfolded to remove the dependencies. In later sections of this paper, we will deal with domains where method override does not make sense, and we will not include it as a feature of the object system.

To handle method interdependence, we include a self–application within a method invocation. In the literature [2, 17], and in many implementations, objects are treated as records, and method invocation is then a composition of field selection and self–application. This gives us an untyped object calculus of the following form:

Let o be the object $\{l_i = \lambda x_i.b_i\}$ (l_i distinct for $i \in I$)
Method selection: $o.l_j = o[l_j](o) = b_j\,[o/x_j]$
Method override: $\begin{aligned} o.l_j &\leftarrow \lambda z.m \\ &= o\,[l_j \leftarrow \lambda z.m] \\ &= \{l_j = \lambda z.m, l_i = \lambda x_i.b_i\} \quad (i \neq j) \end{aligned}$

Let's consider an example that seems common to much of the object–oriented literature: a two dimensional point in $\mathbb{Z} \times \mathbb{Z}$. The point $p = (-1, 7)$ would be the following object:

$$\boxed{p = \{x = \lambda x. -1; y = \lambda x.7\}}$$

The methods x and y are constant methods. Retrieving the x coordinate of the point p is a field selection, plus a self–application:

$$p.x = p[l_x](p) = (\lambda x. -1)\,(p) = -1$$

Now consider the class of movable points *MovablePoint*, which is a *Point* together with a functional method *Move* that produces a new point relative to the original point. The *Move* function computes the new position from existing x and y methods, and overiddes them with new values. The point $p = (-1, 7)$ can be implemented as:

$$\boxed{\begin{aligned} &p = \{ \\ &\qquad x \quad = \lambda s. -1 \\ &\qquad y \quad = \lambda s.7 \\ &\qquad \mathit{Move} = \lambda s, i, j.s\,[x \leftarrow s.x + i]\,[y \leftarrow s.y + j] \\ &\} \end{aligned}}$$

Next, we wish to give the signature type. The obvious signatures are the following:

$$\boxed{\boxed{\begin{aligned} &\mathit{Point} = \{ \\ &\qquad x\colon \mathit{Point} \to \mathbb{Z} \\ &\qquad y\colon \mathit{Point} \to \mathbb{Z} \\ &\} \end{aligned}} \quad \boxed{\begin{aligned} &\mathit{MovablePoint} = \{ \\ &\qquad x\colon \mathit{MovablePoint} \to \mathbb{Z} \\ &\qquad y\colon \mathit{MovablePoint} \to \mathbb{Z} \\ &\qquad \mathit{Move}\colon \mathit{MovablePoint} \to \mathbb{Z} \to \mathbb{Z} \to \mathit{MovablePoint} \\ &\} \end{aligned}}}$$

Clearly, we are going to have a problem with these signatures. Making the types recursive will be a feat because they are anti-monotonic. One of the possible solutions is to hide the negative occurrence of the signature type definition by providing a primitive signature type constructor. Yet, there are still problems deciding when a signature is well–formed, because of the seemingly necessary condition that there be an object belonging to the signature in order to prove well–formedness of the the signature type (due to the self application).

Another technique that will prove very useful is to use a bounded fixed-point $\mu\,(T' \subseteq T.B[T'])$, where

$$\mu\,(T' \subseteq T.B[T']) \equiv \mu\left(T. \bigcap_{T' \subseteq T} B[T']\right).$$

Such fixed points are trivially monotonic. Furthermore, they are useful for expressing polymorphism of the defined types. We will be using technique in our development.

From here we develop a type–theoretical account of the object system. Using the expressive power of the type theory, and the dependent record extension, we can develop a type for signatures in a normal type–theoretic, well–founded manner. Before we proceed with this, we will give a brief discussion the record calculus that we build on top of the very-dependent function type.

3.2 A record calculus

The record calculus we will use to construct objects has a simple encoding on top of the very–dependent function type. A record has a following features: it has a collection of labels, which are usually just names, although arbitrary types are permissable. The record allows access to its fields by name, and it provides an operation to replace a field by new value. This has a natrual encoding using a function type as follows:

Let the record type be **Record** $(A \to B)$ where the domain of labels is A and the domain of values is B. Let $r \in$ **Record** $(A \to B)$	
Record coding:	**Record** $(A \to B) \equiv A \to B$
Field selection:	$r_a \equiv r(a)$ (where $a \in A$)
Field replacement:	$r\,[a \leftarrow b] \equiv \lambda i.\, \mathbf{if}\ i = a\ \mathbf{then}\ b\ \mathbf{else}\ r(i)$ (where $a \in A$ and $b \in B$)

For dependent records, we have a slightly different encoding, because the domain of labels must be well–ordered and the value types depend on their label and the record itself. In this case, the type for dependent records **Record** $\{r\,|a{:}\,A \to B\}$ is the very–dependent type $\{r \mid a{:}\,A \to B\}$. the coding for field select and replacement does not change.

3.3 Types for an object system

At this point we are prepared to give an object system. We will build signatures recursively, one method at a time. The signature for a module will be an incremental construction of partial signatures for the module. Each signature has a collection of *parent* signatures that it inherits methods from, and an additional method type that may depend on the parent signatures as well as the signature being defined. We get a following template for a signature:

Parent function:	*parents*
ParameterType:	*Type*
Method:	*Type*

As we mentioned in section 1.1, we use the template to define a type of *method descriptions*, from which the signatures can be computed. We will collect the method descriptions into a single database, which we will implement as a record that specifies a complete signature. Each method is assigned a unique label in the database record, and the database is ordered so that each template may refer to the descriptions of its parents. For the moment, the labels of the database record are inhabitiants of the type *Atom*, which consists of finite length strings of characters. Given the template for the signature, the database record of the parents f, and a list of parent names, we get the following type for a method description:

$$
\begin{array}{l}
\texttt{type}\ \mathit{MethodDescription}(f) == \{ \\
\qquad \mathit{parents}{:}\ \mathit{Atom\ list} \\
\qquad \mathit{method}{:}\ \mathit{Signature}\ (f, \mathit{parents})) \to \mathit{Type} \\
\}
\end{array}
$$

The *Signature* function computes the signatue of the *parents* from the signature description f. This signature description record has the following form:

$$\texttt{type}\ \mathit{SignatureDescription} == \{f \mid \mathit{name}{:}\ \mathit{Atom} \to \mathit{MethodDescription}(f)\}$$

These two specifications are the key parts of the definition of formal objects. The *MethodDescription* specifies a particular method of an object, and *SignatureDescription* specifies an entire object. The *Signature* function computes the signature from its description. Since we are implementing objects as dependent records, the signature is a dependent record type, which we can build by collecting the ancestors of the object.

```
let Signature f names =
    let Join object domain [] = ⟨object, domain⟩
    and Join object domain (name :: names) =
            if name ∈ domain then
                Join object domain names
            else
                let sig = f(name)
                    names' = names@sig.parents
                    ⟨object', domain'⟩ = Join object domain names'
                in
                    (object' [name ← sig.method (object')] , name :: names')
    and ⟨object, domain⟩ = Join {} []names
    in
        object
end
```

The auxiliary function *Join object domain names* constructs the signature, with the following arguments:

object:	the partially constructed record,
domain:	a list of labels in the partially constructed record,
names:	the labels to be added to the record,
f:	the signature description record

Each method is added only once to the record. When a method is to be added (with name in the *name* argument), it is added only if it isn't in the *domain* list. If it hasn't already been added, its parents are added recursively, and then the *object* record is extended by a new field with key *name* and and type $method(object')$, where *method* is the "method" value in the signature description, and $object'$ is the object that has been extended to include the parent objects.

The *Join* function terminates because the signature description for each object is finite: assign each *name* in the *names* argument to *Join* with metric 0 if the signature description corresponding to *name* has no parents, or the maximum of the parent metrics plus one otherwise. Each description has a finite number of parents, so this maximum exists. To order the *names* arguments, let $names > names'$ if m and m' are the list of metrics for *names* and $names'$, sorted in decreasing order, and m is lexicographically larger than m'. In each call to *Join*, the head *name* in *names* is removed, and possibly replaced by the list of its parents. If *name* has metric n, then it is replaced with a finite number of names with metric $n-1$, respecting the order on the name list. Since no name has metric less than 0, the function must terminate after a finite number of recursive unrollings.

We must still prove that the *Signature* function, and the record types *MethodDescription* and *SignatureDescription* have well–formed types. This uses an inductive argument, similar to the argument for termination of *Join*. In the base case, if an object has no parents, then *Signature* computes to *Unit*, so the *MethodDescription* and *SignatureDescription* for the method are well–formed. Otherwise, assume that the descriptions of the parents of a method have well-formed types. Then *Signature* computes to the join of the record types for the parents, which is a well–formed type, and so the *SignatureDescriotpn* and *SignatureDescription* for that object are well–formed.

This argument has been partially carried out formally. It is a difficult formal proof because of the recursive nature of the description and the inductive argument that is required. The proof is fairly easy for a modification to this system, where multiple inheritance is allowed, but common ancestors are not joined. Although the proof of the modified system is fairly easy, multiple inheritance is a fundamental feature that we wish to support, *with* the joining of common ancestors. The extended proof is still partial.

These two types, *MethodDescription* and *SignatureDescription*, and the function *Signature* are the main tools that we will use to implement the object system. In the next section we show how to implement a standard view of an object system using these tools.

3.4 Polymorphism and record extension

We discuss a slight modification to the object–system before we proceed, to deal with polymorphism and record extensions. Generally, methods will be polymorphic over all subtypes of the signature that the method is specified in. We wish to capture this polymorphism in the type system. Consider the type *MovablePoint* that we described in section 3.1:

$$
\boxed{\begin{array}{l}
\mathit{MovablePoint} = \{ \\
\quad x\colon \mathit{MovablePoint} \to \mathbb{Z} \\
\quad y\colon \mathit{MovablePoint} \to \mathbb{Z} \\
\quad \mathit{Move}\colon \mathit{MovablePoint} \to \mathbb{Z} \to \mathbb{Z} \to \mathit{MovablePoint} \\
\}
\end{array}}
$$

Using the incremental construction of the previous section, this signature would be composed of three sub–signatures:

$\mathit{MovablePoint}_x$ `is` $\{\}$ **method** $x\colon \mathit{MovablePoint}_x \to \mathbb{Z}$	$\mathit{MovablePoint}_y$ `is` $\{\}$ **method** $y\colon \mathit{MovablePoint}_y \to \mathbb{Z}$
$\mathit{MovablePoint}$ `is` $\{\mathit{MovablePoint}_x, \mathit{MovablePoint}_y\}$ **method** $\mathit{Move}\colon \mathit{MovablePoint} \to \mathbb{Z} \to \mathbb{Z} \to \mathit{MovablePoint}$	

In each of the sub–signatures, the method type refers to its enclosing signature. Since the type definition is anti–monotonic in the signature, we would not expect a normal fixed–point construction. The types in this example do not in fact need to use a self–reference. The arguments to the methods x and y can have any type, and the *Move* method is polymorphic over all records having integer methods x and y. In general, we will allow a method to override itself. If a method refers to itself, it should be polymorphic over subtypes of its own signature, or if not, it should be polymorphic over subtypes of the signatures of its parents.

Of course, even a method that overrides itself need not refer to itself using the "self" argument. If it wishes to refer to itself, it can use a standard fix-point, or Y–combinator construction. Thus the "self" argument to a method can always have the type that is the join of its parent signatures, and we never need consider an anti–monotonic recursive construction. However, monotonic definitions may appear more useful. For instance, consider non–empty lists of *Point*s, which might have the following signature:

Point_x `is` $\{\}$ **method** $x\colon \mathit{Top} \to \mathbb{Z}$	Point_y `is` $\{\}$ **method** $y\colon \mathit{Top} \to \mathbb{Z}$
Point `is` $\{\mathit{Point}_x, \mathit{Point}_y\}$ **method** $\mathit{void}\colon \mathit{Top} \to \mathit{Unit}$	
$\mathit{PointList}_{tail}$ `is` $\{\mathit{Point}\}$ **method** $\mathit{Tail}\colon \mathit{Top} \to \mathit{PointList}_{tail}$	
$\mathit{PointList}$ `is` $\{\mathit{PointList}_{tail}\}$ **method** $\mathit{Cons}\colon \bigcap_{T \subset \mathit{Point}} \bigcap_{T' \subset \mathit{PointList}_{tail}} T \to T' \to T'$	

In this signature, the *Tail* method is either $\cdot$ or it is a subtype of *Point*. The *Cons* method adds a point to a list. Even in this case, it is possible to weaken the type of *Tail* to $\mathit{Top} \to \mathit{Point}$ because the *Cons* method has a more constrained type. In practice, we may find that a recursive formulation is never needed.

On the other hand, let's consider a recursive object system. We will maintain the structure of the objects of the last section, and modify the types to account for the recursion. The subtypes that we need are just the *record extensions*. A record extension for a record r is a record that has more fields than r, but with the same type of labels. A general subtype of r allows the label types to grow. An extension of a signature is a signature with more methods, but the same type of method labels. For each signature extension there is a corresponding signature *description* extension. Our recursive sobject system will be based on extended signature descriptions.

The method types depend on polymorphism over record extensions, so we must allow methods to use extended descriptions to compute their type. The new *MethodDescription* is:

$$
\boxed{\begin{array}{l}
\texttt{type}\ \mathit{MethodDescription}(f, T) == \{ \\
\quad \mathit{parents}\colon \mathit{Atomlist} \\
\quad \mathit{method}\colon \mathit{Signature}\ (f, \mathit{parents})) \to T \to \mathit{Type} \\
\}
\end{array}}
$$

Where f is a *SignatureDescription* of the join of the parents, as before, and T is the type of all record extensions of the signature description of this method.

The new *SignatureDescription* requires a bounded, recursive type construction.

$$\boxed{\texttt{type}\ \mathit{SignatureDescription} == \mu\,(T \subseteq S.\,\{f \mid \mathit{name}\colon \mathit{Atom} \to \mathit{MethodDescription}(f, T)\})}$$

The *Signature* function changes slightly, as it must provide a signature description to methods, in order to compute their type.

This new form of recursive type is slightly more dubious than the last. We haven't attempted a formal proof, and although the intuition here seems correct, it not not clear that that we can produce a well–formed type. This is an interesting area for further research, although we can continue with an object system that does not involve recursion. In the next section we discuss constructors for signature and method descriptions.

3.5 Object system

We can develop a standard collection of constructors for signatures and object descriptions that give a more elegant notation to their descriptions. We would like to have the following features:

1. Each method type is defined, it is defined in a context where subtypes of the ancestors are represented by their names. The *Self* type is understood.

2. A signature is automatically constructed at the time a method is defined.

We can perform these operations formally within the system because signature descriptions are formal objects. When a method is defined, it is possible to walk the signature descriptions of the parents, join them, and extract the signature names for each of the ancestors. The new method can be defined in the scope where the ancestor names are bound to the intersection of their record extensions.

To define a new signature with a method, we define the followng operation.

NewSignature $\langle \mathit{name} \rangle$ **inherits** $\langle \mathit{parentlist} \rangle$ **method** $\langle \mathit{name} \rangle : \langle \mathit{method\text{–}type} \rangle$

If the **inherits** field is omitted, then there are no ancestor signatures, and if the **method** field is omitted, it defaults to the identity function. Using this template, we can define the *MovablePoint* example as follows:

NewSignature Point_x **method**$x\colon \mathbb{Z}$	**NewSignature** Point_y **method**$y\colon \mathbb{Z}$
NewSignature *Point* **inherits** $[\mathit{Point}_x; \mathit{Point}_y]$	
NewSignature *MovablePoint* **inherits** $[\mathit{Point}]$ **method** $\mathit{move}\colon \mathit{Point} \to \mathbb{Z} \to \mathbb{Z} \to \mathit{Point}$	

The monoid example of section 1.1, is specified in Figure 3.

3.6 Propositional meaning of signatures

As we will see in the next section of this paper, object signatures have an interesting Curry–Howard interpretation as propostions. The method types of the signatures are *axioms* of the signature, where the object is an implementation of those axioms. The implementation of an object may further extend the signature with theorems that are derivable fro its axioms. Within the object, the axioms are general properties that hold, and the theorems can be trated as axioms that have been proven. In other words, a theorem is a method that has a default implementation provided. As we would expect, further refinements of may override the implementation provided by the theorem with a different, possibly more efficient value.

NewSignature $Monoid_{car}$
method car: $Type$

NewSignature $Monoid_{=}$
inherits $[Monoid_{\oplus}]$
method $=: car \to car \to Type$

NewSignature $Monoid_{eq-ref}$
inherits $[Monoid_{=}]$
method $eq\text{–}prop_{ref}: \forall x: car.x = x$

NewSignature $Monoid_{eq-sym}$
inherits $[Monoid_{=}]$
method $eq\text{–}prop_{sym}$:
$\forall x, y: car.x = y \Rightarrow y = x$

NewSignature $Monoid_{eq-trans}$
inherits $[Monoid_{=}]$
method $eq\text{–}prop_{trans}$:
$\forall x, y, z: car.x = y \Rightarrow y = z \Rightarrow x = z$

NewSignature $Monoid_{\oplus}$
inherits $[Monoid_{\oplus}]$
method $\oplus: car \to car \to car$

NewSignature $Monoid_{op-assoc}$
inherits $[Monoid_{\oplus}]$
method $assoc\text{–}prop$:
$\forall x, y, z: car.(x \oplus y) \oplus z = x \oplus (y \oplus z)$

NewSignature $Monoid_{e}$
inherits $[Monoid_{\oplus}]$
method $e: car$

NewSignature $Monoid_{identity}$
inherits $[Monoid_{=}; Monoid_{\oplus}; Monoid_{e}]$
method $identity\text{–}prop: \forall x: car.x \oplus 1 = x$

NewSignature $Monoid$
inherits $[Monoid_{identity}; Monoid_{eq-ref}; Monoid_{eq-sym}; Monoid_{eq-trans}]$

Figure 3: Signature for a monoid

This suggest a model where the construction of signatures proceeds in the same way that a theory is constructed. The axioms of the theory are stated, and then theorems are proved based on the axioms. The signature of the theory is record consisting of its axioms, and an object of the theory is a record of proofs of the axioms. Thus we have two models for the object system, one is the standard algebraic model of abstract data types (with re–use), and the other is a logical model based upon *theories* or *book* of the logic. These are not competing views; they are complimentary, and within the standard type theory the views are isomorphic, and we can compute one from the other.

4 Type theory and "theories"

In the preceeding section, we saw how to describe objects in the type theory, and we noted their similarity to "theories" within the type theory. There is one difference that we explore in the following sections. Some *theories* assume that their axioms are not only true, but they are well–formed. The type theory itself is an instance of this case. The axioms are proved true and well–formed according to some meta–theory, and it often doesn't even make sense to prove that an axiom is well–formed. There is no type for these theories. In this section, we show how the meta–type theory can be extended, without modifying the semantics, so that theories will correspond to meta–objects, and the meaning of an object corresponds to the meaning of the theory, according to the Curry–Howard correspondence.

4.1 Type theory foundations

Let's consider the fundamental components of a *constructive* type theory. The type theory usually assumes some sort of functional computation system (most commonly the λ-calculus). The type theory does not usually distinguish between β-equivalent terms in the computation system (although there may be special-purpose systems with finer granularity that deal with computational complexity classes of terms). The computational terms exist *before* the types exist.

The types are added to the theory next, in an effort to classify the computational terms. The computational terms exist regardless of whether they have a type or not, but the type system imposes a meaning on the terms. For instance, some terms may abstract a computation over an argument (as a λ abstraction). The type system classifies some of these, and calls them *functions*.

In a constructive type theory, the type system may impose an additional requirement that the types denote predicates in a logic. This is the *Curry-Howard* isomorphism, and it describes a correspondence between the terms of a type and the proofs of a proposition.

Next, the type theory contains a binary equality, coarser then the computational equality, to describe what it means for two terms to be equal, or for two types to be equal.

Finally, the type theory contains rules, which are axioms that partially describe when term can be classified with a type, or when two terms or types are equal. A rules consists of *sequents*,

$$\overline{x}\colon \overline{H} \vdash \overline{T} \quad j\colon \mathtt{ext}\,(\overline{x})\,,$$

where $\overline{x}\colon \overline{H}$ is a dependent record (telescope) or types, $\overline{T}$ is a collection of types. The elements $x\colon H$ left of the turnstile are called *hypotheses*, and the types $\overline{T}$ right of the turnstile are called *conclusions*. Let I be the index set of the hypotheses, with well-founded order $<_I$, and let J be the index set of the conclusions, with well-founded order $<_J$. Then the *meaning* of the sequent is roughly as follows:

- *Assume* for each $x_i\colon H_i$ where $i \in I$, that H_i is a type, inhabited by a term denoted by x_i, under the same assumption for all $x_k\colon H_k$ where $k <_I i$.

- Then, one of T_j for $j \in J$ is inhabited by the term $\mathtt{ext}\,(\overline{x})$ (identically, T_j is true with proof corresponding to $\mathtt{ext}\,(\overline{x})$).

The sequent is a *subtype* relation, stating that if we have terms inhabiting the hypotheses, than there is an algorithm for generating a term inhabiting a conclusion. Since the algorithm decides which conclusion is true, the rules are usually reduced to form involving a single conclusion.

The NuPRL logic refines the meaning of the sequent even more. In this case, the meaning of the sequent describes the equality of a type as follows:

- *Assume* for each $x_i\colon H_i$ where $i \in I$, that $H_i[\overline{x_k}]$ is true, where $\overline{x_k}$ is the vector of all x_k for $k < i$, and $H_i]\overline{x_k}] = H_i[\overline{x_k}']$ for $\overline{x_k} = \overline{x_k'}$ according the the types $\overline{H_k}$.

- Then, one of $T_j[\overline{x}]$ for $j \in J$ is inhabited by the term $\mathtt{ext}\,(\overline{x})$, and $T_j[\overline{x}] = T_j[\overline{x}']$ for equal $\overline{x} = \overline{x}'$ according to the assumptions $\overline{H}$.

Thus, the sequent refines the equality of the type T_j, and $\mathtt{ext}\,(\overline{x})$ is an algorithm for generating a term in T_j given terms $\overline{x} \in \overline{H}$.

A rule reflects the sequent onto itself; a rule has the form

$$\overline{x}\colon \overline{S} \vdash \overline{G} \quad j\colon \mathtt{ext}\,(\overline{x})\,,$$

where $\overline{x}\colon \overline{S}$ and $\overline{G}$ are vectors of sequents. The $\overline{x}\colon \overline{S}$ are called *subgoals*, and $\overline{G}$ are called *goals*, terminology taken from backward-chaining theorem provers. Just as for the sequent, there is no increase in generality in having more than one subgoal, so the goal is usually reduced to a single instance, and the rule has the form,

$$\overline{x}\colon \overline{S} \vdash G \quad \mathtt{ext}\,(\overline{x})\,.$$

The *meaning* of the rule is different from the sequent; the rule is a simple proof-theoretical fact. In the usual case, there are a finite number of subgoals, and each subgoal is distinct from the other subgoals (each subgoal is necessarily closed). The rule is a simple statement of the following form:

> If each subgoal is true, the the goal is true, and $\mathtt{ext}\,(\overline{x})$ inhabits its conclusion.

There is no need to ascribe a deeper meaning to the rule, because the deeper meaning of the sequent already provides the semantic foundation for the type theory.

The standard account of the type theory stops here. Any system implementing the type theory has the following organization:

1. there is a fundamental computation system,

2. there is a standard collection of rules that define the type theory,

3. everything else is either definition or a theorem, in a global arena provided by the type theory.

Yet, an alternative account seems natural, especially if we wish to work with more than one type theory:

1. there is a fundamental computation system,
2. for each type theory,
   (a) there is a set of rules defining the type theory,
   (b) everything else is a definition or theorem based on the rules.

In fact, each type theory is like a rule: it assumes a collection of rules, and then proves a collection of theorems based on those rules. We can account for this extended notion of a type theory by generalizing rules. As a new definition, we generalize a sequent recursively. The class of sequents is described by S by the following grammar:

$$\begin{array}{rcl} S & ::= & \mathit{Term} \\ & | & \overline{x}{:}\,\overline{S} \vdash S \quad \texttt{ext}\,() \end{array}$$

The meaning of closed sequents $s \in S$ is as follows:

- If $s \in \mathit{Term}$, then s denotes a type that is inhabited.
- Otherwise s is non-trivial sequent. Let the index set of the hypotheses be I with well-founded order $<_I$, and let
$$s \equiv \overline{x_i}{:}\,\overline{s_i} \vdash g.$$
  Assume the following, for each $x_i{:}\,s_i$:
  - if $s_i \in \overline{y}{:}\,\overline{S} \vdash G$, then s_i is true, and x_i is an untyped term standing for the `ext` term of the sequent.
  - if $s_i \in \mathit{Term}$, then s_i is a type inhabited by the term x_i. Furthermore, s_i is functional over $x_j{:}\,s_j$ for $j <_I i$. If s_j is not a term, this functionality is trivial, since x_j is a unique term. Otherwise, for $x_j = x_j'$ in s_j, functionality implies $s_i(x_j) = s_i(x_j')$.

  Under these assumptions, g is a type inhabited by $\texttt{ext}\,(\overline{x})$. Furthermore, g is functional, so that for $\overline{x} = \overline{x'}$ inhabiting the hypotheses, $\texttt{ext}\,(\overline{x}) = \texttt{ext}\,(\overline{x}')$ in the type g.

The argument that this is a sensible semantics for the generalized sequent is left for further study. For now, we just accept this semantics as the definition of the type system.

The generalized sequent can be used as a basis for all statements within the type theory. In the following sections, we cover templates for sequents (which are the standard forms of assertions), and then we explore the objects of the type theory, and we show their representations a generalized sequents. The type of "objects" will also be a form of generalized sequent.

4.2 Sequent templates and proofs

Using generalized sequents as a basis for the type theory can easily get out of hand. For example, a natural rule of a type theory is probably the AXIOM rule, which can be described as follows:

$$\overline{a}{:}\,\overline{A}, b{:}\,B[a], \overline{c}{:}\,\overline{C[a,b]} \vdash B[a] \quad \texttt{ext}\,(b)\,.$$

This form states that if the goal is identical to a hypothesis, the goal is proved by the term inhabiting the hypothesis. This seems like a natural rule, but if we are restricted to generalized sequents, we require an infinite number of sequents to define the rule, one for each instantiation of $\overline{A}$, B, and $\overline{C}$. Thus, the normal method of expressing rules is by using templates, like the one above. The variables $\overline{A}$, B, and $\overline{C}$ are slots that may be instantiated with arbitrary sequents, and the variables $\overline{a}$, b, and $\overline{c}$ are slots that can be instantiated with arbitrary variables. This method is used throughout the theory, and statements are usually made in the form of templates. It is understood that the templates are forms that are not part of the type theory, but their instantiations are.

If we examine this concept carefully, there seems to be some sort of impredicativity here. For instance, is the following sequent true?

$$\left[\overline{a}{:}\,\overline{A}, b{:}\,B[a] \vdash B[a] \quad \texttt{ext}\,(b)\right], i{:}\,\mathbb{Z} \vdash \mathbb{Z} \quad \texttt{ext}\,(i)$$

At first glance it seems that the sequent is valid, but upon further reflection, we see that the sequent can't be instantiated to a true sequent. The template assumption can't be "applied" to itself. Instead of the previous sequent, we should be proving the following:

$$\left[\overline{a}: \overline{A}, b: B[a] \vdash B[a]\, \texttt{ext}\ (b) \vdash\right] [i: \mathbb{Z} \vdash \mathbb{Z}\, \texttt{ext}\, (i) \quad],$$

which has no problems with self-application.

Another form of description that is often used is a *tactic*, which is a program that can be used to generate a sequent. In a backward-chaining system, the tactic is given the goal sequent as an argument, and it computes subgoals that are sufficient to prove the goal, along with a program that computes the term inhabiting the goal. Tactics are usually arbitrary programs, and can't be trusted to produce only true sequents. Thus, in addition to the subgoals, the tactic is required to compute a verification, which is essentially a proof tree where the steps of the proofs are all assumptions of the theorem.

But what is a proof? In order to define it, we need at least *one* proof rule. The choice we make is, as would be expected, a version of *modus–ponens*:

$$\frac{\overline{c}: \overline{C}, \overline{a_j}: \overline{A_j} \vdash A_i \quad \texttt{ext}\,(a_i)^*}{\overline{c}: \overline{C}, \left[\overline{a}: \overline{A} \vdash B \quad \texttt{ext}\,(b) \vdash\right] B \quad \texttt{ext}\,(b)}$$

: Let I be the index set of $\overline{A}$ with order $<_I$. Then there are I sequents, one for each $i \in I$, with j ranging over all I where $j <_I i$.

Note that if $\overline{A}$ is empty, the proof terminates.

It is important to note the interaction between proofs and sequent templates. In general, the template stands for an infinite number of sequents. Thus it would seems that modus–ponens would generate an infinite number of subgoals. This would be a significant practical problem! But it is not necessary if an application of modus–ponens keeps sequents in template form, instantiating only those parts of the sequent that are refined by the goal. In essence, an application of modus–ponens requires a unification of the applied sequent to the goal, and the most general unifier is used to create the subgoals.

Of course, the use of sequent templates may still cause problems if an use of the sequent generates an infinite term corresponding to the proof. For instance, suppose we have a thinning rule:

$$\left[\overline{a}: \overline{A}, \overline{c}: \overline{C} \vdash D \quad \texttt{ext}\,(d)\right] \vdash \left[\overline{a}: \overline{A}, b: B, \overline{c}: \overline{C} \vdash D \quad \texttt{ext}\,(d)\right]$$

If we restrict B to match a single hypothesis, not a pattern, then an infinite number of applications of thinning would be required to thin a single template. If the proof term is modified in the operations, the resulting proof term may be infinite. This becomes another task of the type theory designer.

4.3 Objects of the type theory

In general, statements in a type theory are assertions under assumptions. The one assumption present in every statement of the type theory is the underlying computation system. We can specify the computation system in the type theory, but we must still assume that the computation system obeys its specification. For any particular type theory, the basic rules of the theory are presented next, under no additional assumptions. The rules themselves are assertions, and state their assumptions explicitly. To a user of the type theory, the rules and computation system form a basic assumption present in every statement that is made. The type theory is not usually listed in each theorem, but it forms an implicit assumption.

A theorem itself is a degenerate form of a sequent. If the theorem proves a proposition T, it is expressed as the sequent $\vdash T\, \texttt{ext}\,(t)$ where t is the term corresponding to the proof of t. If the assumptions are listed explicitly, the sequent would have the form $\overline{R} \vdash T\, \texttt{ext}\,(t)$ where $\overline{R}$ is the list of rules of the type theory. This sequent states that T is a type of the type theory, and it is inhabited by the term t.

As we noted before, *tactics* also represent a collection of sequents, and we could list the tactics as assumptions of the theorem. Doing so is not necessary, since the tactics provide no extra information, but we may wish to do so provides tactics that are *local* to a theorem. If tactics are written formally and verified, it is possible to treat a tactic as another, derived, instance of a sequent.

Every instance of an assertion in the logic is a generalized sequent, and each true generalized sequent is a *theory*. The axioms of the theory are listed to the left of the turnstile, and the theorems are listed to the right of the turnstile. In general, a theory does not have a type, but some subparts of the theory may have a type. If the parts of a sequent at typable, under a collection of typing assumptions, then there is a type equivalent to the sequent. This type will correspond to a signature, an instance of the axioms will be an object, and the sequent itself is a theorem proving its conclusions.

When a sequent is not typable, there is no corresponding signature. We can still view the sequent as the specification for an object, and some of the operations that make sense on objects, makes sense in the theory. For instance, method selection corresponds to the application of an axiom. Some operations do not make sense. for instance, method override in this setting can only replace non–typable assumptions with equal terms, which is no override at all. However, the general concept of the object system applies in this setting, and it is possible to compute the object view of a theory from its sequent, and vice–versa.

By noting this correspondence, we find that there is only one object in a type theory: a generalized sequent. There are several views: a sequent can stand for a rule of the logic, a theorem, a theory, or a signature. A tactic is a collection of sequents that can be computed. An extract of a sequent is an extract of a rule, a proof of a theorem, or a method implementation in an object. The view that a user takes of an object is a matter of preference; all views have the same underlying meaning.

5 related-work

There has been widespread research into formal methods for object–oriented programming. Abadi and Cardelli [2, 1] have defined several typed and untyped object calculi. They demonstrate how hard it is to find a semantics for an object–calculus, and they eventually use an axiomatic untyped semantics. They also model the method selection as a combination of field selection and self–application. In their work, they define a type *Self* that stands for the signature of the object being defined, and in order to type an object they must assume that the type is well–defined.

Mitchell, Honsell, and Fisher [17] develop an object calculus based on the untyped lambda calculus, and develop a static type checking system to later develop equational reasoning about method bodies. Their type system allows methods to be specialized as they are inherited, by polymorphism over subtypes.

Much of the basic theory of object–oriented programming is based on record calculi, and the semantics of subtyping and record extension. Amadio and Cardelli [3] discuss subtyping in the presence of recursion. In general, recursive subtyping produces a type larger than simple "record extension." Harper and Pierce [11] present a record calculus, and they apply some of their work to an ML–style module system [10]. Wand [20, 21] consider type–inference in the setting of record extension and multiple inheritance, a property that we do not discuss in this paper, but is a useful practical tool. Breu [6] applies the techiques of algebraic specification to objects and their signatures. Dybjer has produced results on dependent inductive types with a set theoretic semantics.

There is a wealth of information on formal object systems in Gunter and Mitchell [9]. Rémy, Wand, and Ohori and Buneman develop type inference in object systems. Cardelli, Mitchell, and Rémy develop record calculi. Kamin and Reddy give semantics models of inheritance in record calculi, and Cook, Hill and and Canning decouple inheritance and subtyping, as in a language like Eiffel [16].

There is relatively less work on an interpretation of *theories.* Basic and Constable [5] discuss logical frameworks. They present a meta–logic that is similar our presentation of theories in 4. In their framework, the language of the logic and its proofs are formal objects belonging to abstract data types in *some* meta–type–theory. In our account, we reduce this meta–type–theory to two components: the generalized sequent and modus–ponens. Pitts [19] presents a view of theories as *categories.* In many ways, the categorical view is similar to an object approach, where signatures are viewed as categories.

Jackson [12] has provided a rich formal environment in the NuPRL system [7] and applied it to constructive abstract algebra. Jsckson's work provides the grounds for this proposal—due to the power of his system, it is possible to formalize large areas of mathematics. In doing so, we become aware that the domains we are formalizing are themselves objects that can be described formally. In a poor environment, where formalism is small, the issue would never arise. Our work is also an extension of Jackson's formalization of abstract algebra.

6 Conclusion

We have presented a formal account of abstract data types with inheritance (an object–oriented view) on top of a type theory. While much of our work required extending the theory with "very–dependent" types, such as a dependent record type, the expressivity and semantics of the existing type theory presented a powerful framework in which to give the formal account. Some of the problems that exist in other accounts of object–oriented programming have standard solutions in the type theory (for instance, dependent method typing). We have yet to find an answer for other problems (for instance, recursive record typing).

The type theory also provides a standard framework of "propositions–as–types," and provides an immediate interpretation of object signatures as theories. In fact, the work developed in this proposal began as a formal account of the informal theories of a system like NuPRL. This view eventually leads us to an interpretation of the entire type theory as a meta–object, where the single component of the system is a seuqent, with various views as a rule, theorem, or meta–signature.

I have developed a basic theory of objects in the NuPRL system. For my thesis, I propose to extend the theory, and develop some of the work presented here to remove some holes and fill in some of the details. I feel that it is important to actually implement a system as proposed. Not only do we get an formal account of a modularity and re–use, we can apply our work to the type theory itself.

References

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second–order systems. In *Proceedings of European Symposium on Programming*, pages 1–24. Springer, April 1994.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first–order systems. *Theoretical Aspects of Computer Software*, pages 296–320, April 1994.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 104–118. ACM, January 1991.

[4] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North–Holland, 1984.

[5] David A. Basin and Robert L. Constable. *Metalogic Frameworks*, pages 1–29. Cambridge University Press, 1993.

[6] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Springer–Verlag, 1991.

[7] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mender, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics in the NuPRL Proof Development Systen.* Prentice–Hall, 1986.

[8] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[9] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object–Oriented Programming.* MIT Press, 1994.

[10] Robert Harper and Mark Lillibridge. A type–theoretic approach to higher–order modules with sharing. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 123–137. ACM, January 1994.

[11] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 131–142. ACM, January 1991.

[12] Paul Bernard JAckson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, January 1995.

[13] Paul Francis Mandler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, September 1987. 87–870.

[14] Per Martin–Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North–Holland, 1975.

[15] Per Martin–Löf. *Intutionistic Type Theory.* Bibliopolis, Napoli, 1984.

[16] Bertrand Meyer. *Object–Oriented Software Construction.* Prentice–Hall, 1988.

[17] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *IEEE Symposium on Logic in Computer Science*, 1993.

[18] Christine Paulin-Mohring. Inductive Definitions in the systen Coq.

[19] Andrew Pitts. *Categorical Logic*, volume VI. Oxford University Press, 1995. (forthcoming).

[20] Mitchell Wand. Complete Type Inference for Simple Objects. In *Symposium on Logic in Computer Science*, pages 37–44. IEEE, June 1987.

[21] Mitchell Wand. Type Innference for Record Concatenation and Multiple Inheritance. In *Fourth Annual Symposium on Logic in Computer Science*, pages 92–97. IEEE, June 1989.

A NuPRL Type Theory

In this Appendix, we discuss the NuPRL type theory in some detail. By doing so, we hope to justify the inductive extsion to the type theory of section 2.

A.1 NuPRL Types

The core of the type theory is based upon the work by Martin–Löf in [15, 14]. We discuss the basic attributes of the type theory here. For further information the reader is referred to Martin–Löf's original paper, and to the base NuPRL document [7]. For a complete, formal account of the type theory with its semantics, see Stuart Allen [?].

There are several *base* types in the type theory, including

- *Void*: the empty type,
- *Atom*: the type of finite strings of characters,
- $\mathbb{Z}$: the integers,
- an order $i < j$ which is a type for any two integers i and j,
- a ternary equality, of the form $x = y \in T$ where T is a type, and x and y are objects of type T,
- the types of types $\mathbb{U}_i$, where a type in $\mathbb{U}_i$ also belongs to $\mathbb{U}_{i+1}$.

If an equality type is inhabited, it is solely inhabited by the term "$\cdot$". The *Unit* type, which is solely inhabited by the term $\cdot$, is defined as $0 = 0 \in \mathbb{Z}$. Similarly, the empty type *Void* could be defined as $0 = 1 \in \mathbb{Z}$, which is provably false and is not inhabited. However, the *Void* type is frequently used, and was found to be more convenient to define it as a primitive. Using the propositions-as-types correspondence, *True* can be defined as *Unit*, and *False* as *Void*. Similarly, if the order is inhabited, it contains the single object $\cdot$.

The types $\mathbb{U}_i$ are indexed over $\mathbb{N}$, and form a type hierarchy of *universes* that are monotonically increasing in type inclusion. The NuPRL type theory is *predicative*, as is the standard Martin–Löf type theory.

There are several type constructors that are used to construct composite types. Given two types T_1 and T_2, the *disjoint union* type can be constructed as $T_u \equiv T_1 \mid T_2$. Inhabitants of the disjoint union are tagged according to which branch of the union they belong to. If $x \in T_1$, then $inl(x) \in T_1 \mid T_2$, and if $y \in T_2$ then $inr(x) \in T_1 \mid T_2$. This corresponds to the constructive disjunction $T_1 \vee T_2$.

The dependent product provides a pairing construction similar to a Cartesian product. The term $x{:}\,T_1 \times T_2$ is a type if T_1 is a type, and T_2 is a type for any inhabitant x of T_1. The inhabitants of the product are pairs $\langle x, y\rangle$ where $x \in T_1$, and $y \in T_2(x)$. The dependency serves two purposes. First of all, it is possible to give more precise specifications of some constructive objects. For instance, we may represent a date by its year, its month, and the day of the month, so that the date would inhabit the type $\mathbb{N} \times \{1 \ldots 11\} \times \{1 \ldots 31\}$. However, obviously illegal dates would inhabit this type (February 31, for instance). We can give a more precise specification using the dependent type $\mathbb{N} \times month{:}\,\{1 \ldots 11\} \times \{1 \ldots days\_per\_month(month)\}$, where $days\_per\_month$ is a function with the obvious definition.

Another use of the dependent type is for existential quantification. The type $x{:}\,T_1 \times T_2$ corresponds to the quantification $\exists x{:}\,T_1.T_2$. If T_2 does not depend on x, we get the conjunction $T_1 \wedge T_2$.

A variant of the product type is the *set* type, which is denoted by $\{x{:}\,T_1 \mid T_2\}$. This corresponds to the type $x{:}\,T_1 \times T_2$, except that the inhabitants of the set type are elements of T_1; the inhabitant of T_2 is discarded. However, $T_2(x)$ must have *some* inhabitant for any element $x \in \{x{:}\,T_1 \mid T_2\}$. That is, the type $\{x{:}\,T_1 \mid T_2\}$ is the subtype of T_1 where T_2 holds on every element. In certain parts of a proof it may be possible to use the fact that T_2 holds without needing the inhabitant (for instance, when an equality is being proved). In other cases it is possible to derive the inhabitant, for instance for an equality or order relation (the inhabitant is always $\cdot$). One of the immediate uses of the set to is to express subranges of the integers:

$$
\begin{array}{rcl}
\{i \ldots j\} & \equiv & \{k{:}\,\mathbb{Z} \mid i \le k \wedge k \le j\} \\
\{i \ldots\} & \equiv & \{j{:}\,\mathbb{Z} \mid i \le j\} \\
\{\ldots i\} & \equiv & \{j{:}\,\mathbb{Z} \mid j \le i\}
\end{array}
$$

Another important type constructor is the dependent function type. If T_1 is a type, and T_2 is a type given $x \in T_1$, then $x{:}\,T_1 \to T_2$ is a type that corresponds roughly to the function space with domain T_1 and range $T_2(x)$ for argument $x \in T_1$. Again the dependencies allow functions to be more precisely specified than the undependent type. For instance, all functions that return a day-of-month given a month as a argument belong to $month{:}\,\{1 \ldots 11\} \to \{1 \ldots days\_per\_month(month)\}$. According to propositions-as-types, the function type corresponds to universal quantification. The function type above corresponds to the quantification $\forall x{:}\,T_1.T_2$. If T_2 does not depend on x, then this becomes the normal implication $T_1 \Rightarrow T_2$.

The final type constructor we will discuss is the recursive type definition. For a complete exposition of the constructor, see Paul Mendler's thesis [13]. If $T(X)$ is a type for all types X and if T is *monotonic* in X, then $\mu\,(X.T)$ is a type, and it has a *fixed point* $\mu\,(X.T) = T(\mu\,(X.T))$. The parameterized version is $\mu\,(y, X.T; z)$, which is a type if there is some type Z where z inhabits Z, and for any inhabitant y of Z, and type function $X{:}\,Z \to \mathbb{U}_i$, T is a type that is monotonic in X.

The inhabitants of the recursive types are those elements that inhabit the finite unrollings of the type. For instance, the list of ascending integers described in the introduction has type $L_I(0)$ where

$$L_I(k) \equiv \mu\,(i, X.\,Unit \mid j{:}\,\{i \ldots\} \times X(j+1); k)$$

This type is inhabited by $inl\,\cdot$, since $inl\,\cdot \in Unit \mid j{:}\,\{i \ldots\} \times L_I(j+1)$ Similarly, we can show $inr\,\langle 5, inl\,\cdot\rangle$ and $inr\,\langle 4, inr\,\langle 5, inl\,\cdot\rangle\rangle$ inhabit the type, but not $inr\,\langle 5, inr\,\langle 4, inl\,\cdot\rangle\rangle$, nor $inr\,\cdot$.

Since the recursive definition has a fixed-point semantics, it is necessary to require that the term defining the type (T in $\mu\,(y, X.T; z)$) be monotonic over the type argument (X). It is also necessary that the definition argument be a well-defined type for any value of the type argument. That is, the type argument does not range over unrolling of the type definition, it ranges over all types, and so it is not possible to impose any structure on inhabitants of the type denoted by the type argument. As mentioned in the introduction, this makes it impossible to define a type that depends on the inhabitant of the type argument in a non-trivial way.

A.2 NuPRL Proof System

When we discuss terms in a type theory like NuPRL it is generally necessary to distinguish between *terms*, which are just notation, and meaningful mathematical objects. For instance, $1 + \mathbb{Z}$ is a term, but is not a meaningful mathematical object, and it does not belong to any type. In NuPRL the discourse about mathematical objects is limited to those objects that can be given a type. The semantics of the type theory is such that any term that can be given a type has a valid semantical meaning and corresponds to some mathematical object. On the other hand, since the type theory is necessarily incomplete, if a term does not have a type it may or may not correspond to a meaningful mathematical object. In a Martin–Löf type theory, it is generally undecidable which terms can be given a type, and what the type is. In NuPRL the decision was made to integrate type-checking with theorem proving. That is, when we prove a theorem, we must also prove that the theorem statement is a type. This is an incremental process, and a large part of the proof rules in NuPRL are concerned with checking the *well-formedness* of terms.

In NuPRL the well-formedness of a term is stated using a ternary equality. The general term for the equality is $t_1 = t_2 \in T$. The semantic meaning of an open equality is that:

1. T is a type,
2. t_1 and t_2 are well-formed elements of type T,
3. and t_1 and t_2 are equal using the equality of type T.

Every type provides an equality. It is possible to define an equality for a type using a *quotient* construction. Otherwise, a default equality is provided by the system. Membership $t \in T$ is stated as $t = t \in T$, which has the meaning that t is well-formed, and it is an element of type T. A peculiar characteristic of the equality is that the term $t_1 = t_2 \in T$ is itself not well-formed unless the first two properties of the equality hold; that is, T must be a type, and t_1 and t_2 must be elements of type T. Currently, there is no way to express non-membership.

A.2.1 Sequent semantics

Theorems in NuPRL are expressed in the form of *sequents*, which have *hypotheses* and *conclusions*. Let $x_1{:}T_1, \ldots, x_n{:}T_n$ be a vector of named types, and let $C_1, \ldots, C_m$ be a vector of types given $x_1{:}T_1, \ldots, x_n{:}T_n$, then $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C_1, \ldots, C_n$ is a sequent. In NuPRL, the sequents are further restricted to have only a single conclusion, which aids in enforcing constructivity.

In the sequent $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$, the variables $x_1, \ldots, x_{i-1}$ are bound in T_i for any i where $1 \le i \le n$. All variables $x_1, \ldots, x_n$ are bound in C.

A closed NuPRL sequent $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$ is *true* if, and only if:

- For each i in the range $1 \le i \le n$, the term T_i is a type, and it is pointwise-functional over $x_1, \ldots, x_{i-1}$, where $x_1, \ldots, x_{i-1}$ inhabit $T_1, \ldots, T_{i-1}$.
- C is a type, it is pointwise-functional over inhabitants $x_1, \ldots, x_n$ of $T_1, \ldots, T_n$.

A Hypothesis T_i is pointwise functional if $T_1, \ldots, T_{i-1}$ are pointwise functional, and given $x_1{:}T_1, \ldots, x_{i-1}{:}T_{i-1}$, and $y_1{:}T_1, \ldots, y_{i-1}{:}T_{i-1}$, where $x_1 = y_1 \in T_1, \ldots, x_{i-1} = y_{i-1} \in T_{i-1}$, then $T_i[x_1, \ldots, x_{i-1}]$ and $T_i[y_1, \ldots, y_{i-1}]$ are equal types. A similar definition applies to the conclusion.

A theorem in NuPRL is a proof of a sequent. An unconditional theorem has no hypotheses. The semantics of pointwise-functionality enforce the mathematical meaning of the theorems. The terms in the sequent have semantical meaning as real mathematical objects. Without this restriction, theorems could only be assumed to state proof theoretic properties about terms, and little underlying mathematical significance could be implied. This mathematical significance has it price in the complexity of proofs, for in addition to proving that a sequent it true, the prover must also verify its functionality.

A.2.2 Rule definitions

The proof rules for NuPRL can be divided into five categories:

1. *Equality* rules are used to prove well-formedness, membership, and equality.
2. *Formation* rules are used to construct inhabitants of a type.
3. *Elimination* rules are used to *eliminate* hypotheses. These usually correspond to a function invocation, a case analysis, or induction on the assumption.

4. *Substitution* rules are used to interchange equal terms.

5. *Computation* rules are used to define β-reduction.

A feature of the NuPRL proof system is the incremental construction of proofs and the programs that they correspond to. Since the NuPRL logic is constructive,[1] every proof corresponds to a program that is a *construction* of the theorem. For instance, a proof of $\mathbb{Z}$ is any integer. In NuPRL the proof terms, called *extracts*, are hidden by the proof system so that users do not have to deal with them explicitly. Since the proof term can be constructed syntactically from the proof tree, each rule in the logic has an extract component that determines how to construct the proof term. Once a theorem is proved, the user can ask for the extract term, and the system will construct it from the proof tree and the rule templates.

Formation rules are used extensively in the interactive construction. To the user, the application of a formation rule is a *refinement* of the goal as a proposition into subgoals. The formation rule construct the program inhabiting the proof behind the scenes. Formation rules are not strictly necessary: to prove a proposition, the user could simply supply an inhabiting program, and prove its membership using the equality rule of the type. However, this approach does incremental construction of programs difficult at best. A main goal of NuPRL is to raise the level of programming to reasoning about specifications, after which the programs can be constructed using the propositions-as-types correspondence. Reasoning about specifications is accomplished as an interactive sequence of refinements, and formations rules play a central part. Of course, since formation rules are redundant, it is necessary to show that they agree with the equality rules for each type.

Each rule has

1. a name by which the rule can be invoked,

2. arguments to the invocation,

3. the rule,

4. a template for constructing the extract.

The rules are typically listed in a backward-chaining format. The goal sequent is listed first with the extract construction, then the name with the arguments of invocation, and finally the subgoals. Proofs in NuPRL can be constructed by a goal-directed search. Given a goal sequent, the user attempts to apply a rule. If the rule is successful (as determined by the *refiner*), the subgoals are presented to be proved in turn. For instance, consider the rule for *lessThanEquality* in Table 6. The *goal* is the sequent $H \vdash i_1 < j_1 = i_2 < j_2 \in \mathbb{Z}$. By applying the rule *lessThanEquality*, the user *refines* the goal to the two *subgoals* $H \vdash i_1 = i_2 \in \mathbb{Z}$ and $H \vdash j_1 = j_2 \in \mathbb{Z}$. Note that this is an intensional notion of equality.

A.2.3 Terms

The terms in NuPRL are manipulated using a *uniform representation* that makes binding and subterms explicit. The general form for any term in NuPRL is

$$\mathit{opname}\,\{p_1\colon s_1,\ldots,p_n\colon s_n\}\,(\overline{v_1}.t_1;\cdots;\overline{v_m}.t_m)\,.$$

More formally, let *STRING* be the set of strings and *VAR* be the set of variables. Then we can define the type of terms to be the recursive type

$$\mathit{Term} \equiv \mathit{STRING} \times \big((\mathit{STRING} \times \mathit{STRING})\mathit{list}\big) \times \big((\mathit{VARlist} \times \mathit{Term})\big)\mathit{list}.$$

Each term has an *operator* that is composed of an operator name, and a list of parameters, which are just pairs of strings. The operator is chosen so as to identify the term in a semantically meaningful way. For instance, the uniform representation of the type $\mathbb{Z}$ is the term $\mathit{int}\,\{\}\,()$, and the uniform representation of the number 3 is the term $\mathit{number}\,\{\$\colon 3\}\,()$. In most cases, the operator name identifies the major category of the operator, and the parameters subdivide the category. This is an implementation decision, so for instance, the numbers could be represented by

[1] Constructivity is not a requirement of the NuPRL type theory—the axiom of excluded middle could be added with no loss of consistency.

the terms $number0\ \{\}\,()\,, number1\ \{\}\,()\,, \ldots$, or by the term $number\ \{\$\colon 0\}\,()\,, number\ \{\$\colon 0\}\,()\,, \ldots$. The latter representation makes the interesting parts of the term more explicit, and is the preferred representation.

Each term also contains a list of bound subterms. In each subterm $\overline{v_i}.b_i$, there is a term b_i and a list of binding variables $\overline{v_i}$ that are bound in b_i. The uniform representation makes no claim as to the meaning of the binding; it is purely syntactical. As an example, the λ abstraction $\lambda v.b$ uses the uniform representation $lambda\ \{\}\,(v.b)$, and the definition of λ specifies the meaning of the components of the term.

In the next few sections, we cover the types in NuPRL and their rules. Along with each type, we give the uniform syntax, which specifies what role the parts of the term have.

A.2.4 Rules for base types

In this section we cover the rules for the base types in NuPRL. Most types have a standard set of rules. There are equality and formation rules for the type in its type universe, there are equality and formation rules for members of the type, there is an elimination rule of the type as a hypothesis, and there is an equality for the the induction combinator if it is defined by an elimination rule.

Uniform representation	
Void	$void\ \{\}\,()$

Void **type** The rules for *Void* are shown in Table 3. The equality and formation rules specify that void is a type. There are no rules for member equality or formation, which agrees with the notion that the *Void* type is empty. The elimination rule is simply *ex-falso-quodlibet*. Any extract is allowed for the elimination rule, which agrees with the notion that this branch of the proof is *non-computational*.

$H \vdash Void = Void \in \mathbb{U}_1$ ext $\cdot$
BY voidEquality

Equality rule

$H \vdash \mathbb{U}_i$ ext $Void$
BY voidFormation

Formation rule

$H, x\colon Void, J \vdash T$ ext
BY voidElimination $assumption\text{-}index\ \{\$i\}$

Elimination rule

Table 3: Rules for *Void*

Uniform representation	
Atom	$atom\ \{\}\,()$
"*name*"	$token\ \{name\colon t\}\,()$

Atom **type** The *Atom* type corresponds to a set of disjoint points, each denoted by a unique string. The member equality and formation rules take an argument token, which is specified with the notation "*$tok*", which is a simple form of type checking. There is no elimination rule because there is no ordering on the elements. The rules for *Atom* are shown in Table 4.

$H \vdash Atom = Atom \in \mathbb{U}_1$ ext $\cdot$
BY atomEquality

Equality rule

$H \vdash \mathbb{U}_i$ ext $Atom$
BY atomFormation

Formation rule

$H \vdash$ "$\$tok$" $=$ "$\$tok$" $\in Atom$ ext $\cdot$
BY tokenEquality

Member equality rule

$H \vdash Atom$ ext "$\$tok$"
BY tokenFormation "$\$tok$"

Member formation rule

Table 4: Rules for *Atom*

$\mathbb{Z}$ **type** The $\mathbb{Z}$ type corresponds to the set of integers. The integers are not defined using the Peano axioms. Instead, the type consists of a set of points, and reasoning about ordering and arithmetic are all accomplished through the decision procedure *arith*. Terms are provided for the standard arithmetic constructions (addition, multiplication, negation, etc.). For example, the term 1 is *not* represented as the successor of 0, but it is possible to prove such statements in arithmetic as $0+1 = 1 \in \mathbb{Z}$ and $0 < 1$ using the decision procedure.

Uniform representation	
$\mathbb{Z}$	$int\{\}()$
i	$number\{i{:}n\}()$
$ind(x; y, z.d; b; y, z.u)$	$ind\{\}(x; y_1, z_1.d; b; y_2, z_2.u)$

Equality rule

$$H \vdash \mathbb{Z} = \mathbb{Z} \in \mathbb{U}_1 \quad \text{ext } \cdot$$
BY intEquality

Formation rule

$$H \vdash \mathbb{U}_i \quad \text{ext } \mathbb{Z}$$
BY intFormation

Member equality rule

$$H \vdash i = j \in \mathbb{Z} \quad \text{ext } \cdot$$
BY arith

Member formation rule

$$H \vdash \mathbb{Z} \quad \text{ext “\$i”}$$
BY numberFormation “$i”

Elimination rule

$$H, x{:}\,\mathbb{Z}, J \vdash T \quad \text{ext } ind(x; y, z.down[\cdot/v]; base; y, z.up[\cdot/v])$$
BY intElimination *assumption–index* {$i} $z\ y\ v$

$$H, x{:}\,\mathbb{Z}, J, y{:}\,\mathbb{Z}, v{:}\, y < 0, z{:}\, T[y+1/x] \vdash T[y/x] \quad \text{ext } down$$
$$H, x{:}\,\mathbb{Z}, J \vdash T[0/x] \quad \text{ext } base$$
$$H, x{:}\,\mathbb{Z}, J, y{:}\,\mathbb{Z}, v{:}\, 0 < y, z{:}\, T[y-1/x] \vdash T[y/x] \quad \text{ext } up$$

Equality rule for *ind*

$$H \vdash \left(\begin{array}{ll} & ind\left(x_1; y_{d_1}, z_{d_1}.down_1; base_1; y_{u_1}, z_{u_1}.up_1\right) \\ = & ind\left(x_2; y_{d_2}, z_{d_2}.down_2; base_2; y_{u_2}, z_{u_2}.up_2\right) \\ \in & T[x_1/z] \end{array}\right) \quad \text{ext } \cdot$$
BY indEquality $z\ T\ x\ y\ w$

$$H \vdash x_1 = y_1 \in \mathbb{Z}$$
$$H, x{:}\,\mathbb{Z}, w{:}\, x < 0, y{:}\, T[x+1/z] \vdash down_1\left[x, y/x_{d_1}, y_{d_1}\right] = down_2\left[x, y/x_{d_2}, y_{d_2}\right] \in T[x/z]$$
$$H \vdash base_1 = base_2 \in T[0/z]$$
$$H, x{:}\,\mathbb{Z}, w{:}\, 0 < x, y{:}\, T[x-1/z] \vdash up_1\left[x, y/x_{u_1}, y_{u_1}\right] = up_2\left[x, y/x_{u_2}, y_{u_2}\right] \in T[x/z]$$

Reduction for the induction combinator

$$\begin{array}{lcll} ind(x; y, z.down; base; y, z.up) & \rightarrow & down[x, ind(x+1; y, z.down; base; y, z.up)/y, z] & \text{if } x < 0 \\ ind(0; y, z.down; base; y, z.up) & \rightarrow & base & \\ ind(x; y, z.down; base; y, z.up) & \rightarrow & up[x, ind(x-1; y, z.down; base; y, z.up)/y, z] & \text{if } x > 0 \end{array}$$

Table 5: Rules for $\mathbb{Z}$

Using this methodology, it is possible to use representations of numbers that are logarithmic in the value of the number, and the *arith* decision procedure can be correspondingly fast. The consequence is that *arith* has yet to be formally verified. Since $\mathbb{Z}$ is a special inductive construction, it also requires reasoning that differs from other recursive constructions in NuPRL, such as lists or more general recursive structures. It is felt that arithmetic is used sufficiently often that the efficiency is worth the cost.

The elimination rule for $\mathbb{Z}$ deserves some discussion because it is the first example of induction. The term $ind(x; y, z.down; base; y, z.up)$ is the integer induction combinator. Given an integer x, this combinator has the following reduction:

$$ind\,(x; y, z.down; base; y, z.up) \rightarrow \begin{cases} down\,[x, ind\,(x+1; y, z.down; base; y, z.up)\,/y, z] & \text{if } x < 0 \\ base & \text{if } x = 0 \\ up\,[x, ind\,(x-1; y, z.down; base; y, z.up)\,/y, z] & \text{if } x > 0 \end{cases}$$

When an integer hypothesis is eliminated, the induction principle is used to refine the sequent, and the extract will be defined by the induction combinator. An equality rule is provided for the induction combinator, and reduction rules. The rules are shown in Table 5.

Uniform representation	
$x < y$	$lt\,\{\}\,(x; y)$

$<$ type The $<$ type is used for the ordering relation on $\mathbb{Z}$. It is a type, and it is inhabited by the single element $\cdot$ if it is true. As with other arithmetic statements on the integers, it is proved by the *arith* decision procedure. The elimination rule corresponds to the induction principle. Since there is a single element inhabiting the type, the induction principle consists of a single base case. Syntactically, the elimination rule simply replaces occurrences of the hypothesis with its value.

$H \vdash i_1 < j_1 = i_2 < j_2 \in \mathbb{U}_1$ ext $\cdot$
BY lessThanEquality
$H \vdash i_1 = i_2 \in \mathbb{Z}$ ext $\cdot$
$H \vdash j_1 = j_2 \in \mathbb{Z}$ ext $\cdot$

Equality rule

$H \vdash \mathbb{U}_i$ ext $a < b$
BY lessThanFormation
$H \vdash \mathbb{Z}$ ext a
$H \vdash \mathbb{Z}$ ext b

Formation rule

$H \vdash \cdot = \cdot \in i < j$ ext $\cdot$
BY lessThanMember
$H \vdash i < j$ ext $\cdot$

Member equality rule

$H \vdash i < jbre$ ext $\cdot$
BY arith

Member formation rule

$H, x{:}\, p < q, J \vdash C$ ext t
BY equalityElimination *assumption–index* $\{i\}$
$H, x{:}\, p < q, J\,[\cdot/x] \vdash C\,[\cdot/x]$ ext t

Elimination rule

Table 6: Rules for $<$

Uniform representation	
$\mathbb{U}_i$	$univ\,\{i{:}\,l\}\,()$

$\mathbb{U}$ type The table for the universe rules is pretty simple because the member equality and formation rules are not included. Instead, the rules are listed on the tables for other types as the equality and formation rules. Furthermore, there is no elimination rule, since each universe is just a set of points, with no order.

$H \vdash \mathbb{U}_j = \mathbb{U}_j \in \mathbb{U}_i$ ext $\cdot$
BY universeEquality

Equality rule

$H \vdash \mathbb{U}_i$ ext $\mathbb{U}_j$
BY universeFormation *level–exp* $\{j\}$

Formation rule

Table 7: Rules for $\mathbb{U}$

Uniform representation	
$x = y \in T$	*equal* $\{\}\,(T; x; y)$

Equality type The rules for equality are also short, as the equalities are specified in the tables for each type. The equality is also a type, inhabited by the single element $\cdot$, and the elimination rule is similar to that for $<$

$H \vdash (x_1 = y_1 \in T_1) = (x_2 = y_2 \in T_2) \in \mathbb{U}_i$ ext $\cdot$
BY equalityEquality
$H \vdash T_1 = T_2 \in \mathbb{U}_i$ ext $\cdot$
$H \vdash x_1 = x_2 \in T_1$ ext $\cdot$
$H \vdash y_1 = y_2 \in T_1$ ext $\cdot$

Equality rule

$H \vdash \mathbb{U}_i$ ext $a = b \in T$
BY equalityFormation T
$H \vdash T = T \in \mathbb{U}_i$
$H \vdash T$ ext a
$H \vdash T$ ext b

Formation rule

$H \vdash \cdot = \cdot \in (t_1 = t_2 \in T)$ ext $\cdot$
BY axiomEquality
$H \vdash t_1 = t_2 \in T$ ext $\cdot$

Member equality

$H, x{:}\, p = q \in A, J \vdash C$ ext t
BY equalityElimination *assumption–index* $\{i\}$
$H, x{:}\, p = q \in A, J\,[\cdot/x] \vdash C\,[\cdot/x]$ ext t

Elimination rule

Table 8: Rules for $=$

A.2.5 Rules for type constructors

Each well-formed construction defined by a type constructor is a type, and it requires rules that define equality and formation on the type and its members. The rules are stated as general properties of the type constructor, and the well-formedness of the construction is defined by the equality rule for the constructor. As we go over the standard type constructors, we will give an semi-formal semantics of the construction that is used to define the type and its members.

Uniform representation	
$x{:}\, A \to B$	*function* $\{\}\,(A; x.B)$
$\lambda v.b$	*lambda* $\{\}\,(v.b)$
$f\,(a)$	*apply* $\{\}\,(f; a)$

Function type The function type is used to denote function spaces, where the domain and range have definable types. The function space is typically written $a{:}\, A \to B$, where A is the domain type, and for any argument $a \in A$, the type $B(a)$ is the range type. To be more specific, $a{:}\, A \to B$ is a type with membership φ if-and-only-if:

1. A is a type with membership α,
2. $B(a)$ is a type with membership β_a,
3. for any f such that $\varphi(f)$ and any a such that $\alpha(a)$, $\beta_a f(a)$.

The elements of this type are simple λ abstractions, whose arguments lie in A, and whose values lie in $B(a)$. In addition to the usual intensional equality on members of the function type, it is consistent to define an extensional equality. Two terms are extensionally equal as functions if they

$H \vdash a_1\colon A_1 \to B_1 = a_2\colon A_2 \to B_2 \in \mathbb{U}_i$ ext $\cdot$
BY functionEquality x
$H \vdash A_1 = A_2 \in \mathbb{U}_i$
$H, x\colon A_1 \vdash B_1\,[x/a_1] = B_2\,[x/a_2] \in \mathbb{U}_i$

Equality rule

$H \vdash \mathbb{U}_i$ ext $a\colon A \to B$
BY functionFormation x a
$H \vdash a = a \in \mathbb{U}_i$
$H, x\colon A \vdash \mathbb{U}_i$ ext B

Formation rule

$H \vdash \lambda a_1.b_1 = \lambda a_2.b_2 \in a\colon A \to B$ ext $\cdot$
BY lambdaEquality *level–exp* $\{i\}$ x
$H \vdash A = A \in \mathbb{U}_i$
$H, x\colon A \vdash b_1\,[x/a_1] = b_2\,[x/a_2] \in B\,[x/a]$

Member equality rule (intensional)

$H \vdash f = g \in x\colon A \to B$ ext t
BY functionExtensionality *level–exp* $\{i\}$ $y\colon C \to D$ $z\colon E \to F$ u
$H, u\colon A \vdash f\,(u) = g\,(u) \in B\,[u/x]$ ext t
$H \vdash A = A \in \mathbb{U}_i$
$H \vdash f = f \in y\colon C \to D$
$H \vdash g = g \in z\colon E \to F$

Member equality rule (extensional)

$H \vdash a\colon A \to B$ ext $\lambda z.b$
BY lambdaFormation *level–exp* $\{i\}$ z
$H \vdash A = A \in \mathbb{U}_i$
$H, z\colon A \vdash B\,[z/a]$ ext b

Member formation rule

Table 9: Rules for dependent function (part 1)

are both functions, and they have equal values when applied to equal arguments in the domain type. The rules have a straightforward implementation from the semantics, shown in Tables 9 and 10.

The undependent function type $A \to B$ can be coded in terms of the dependent function type simple by choosing a variable x that is not free in B, and using the type $x\colon A \to B$. The $NuPRL$ implementation provides a special "blank" variable (which we denote by \$) that can only occur in binding positions. The special variable can *never* be free, and the undependent type is coded as $\$\colon A \to B$.

Although the dependent function type rules are sufficient for proving properties of the dependent type, it is useful to provide an optimized set of rules for the undependent type. These rules are listed in Table 11. The formation rule is optimized for the interpretation of the function type as an implication.

Product type The product type constructor $a\colon A \times B$ defines a cartesian product. The elements of the type are pairs $\langle a, b\rangle$ where $a \in A$ and $b \in B(a)$. More formally, $T \equiv a\colon A \times B$ is a type with membership φ if-and-only-if:

Uniform representation	
$x\colon A \times B$	$\mathit{product}\,\{\}\,(A; x.B)$
$\mathbf{let}\ \langle e, x\rangle = y\ \mathbf{in}\ b$	$\mathit{spread}\,\{\}\,(e; x, y.b)$

1. A is a type with membership α,

2. for any a such that $\alpha(a)$, $B(a)$ is a type with membership β_a,

$H, f\colon x\colon A \to B, J \vdash T \qquad \text{ext } t\,[f(a), \cdot/y]\; v$
BY functionElimination *assumption–index* $\{\$i\}\; a\; y\; v$
$H, f\colon x\colon A \to B, J \vdash a = a \in A$
$H, f\colon x\colon A \to B, J, y\colon B\,[a/x]\,, v\colon y = f(a) \in B\,[a/x] \vdash T \qquad \text{ext } t$

Elimination rule

$H \vdash f_1\,(a_1) = f_2\,(a_2) \in B\,[a_1/x] \qquad \text{ext } \cdot$
BY applyEquality $x\colon A \to B$
$H \vdash f_1 = f_2 \in x\colon A \to B$
$H \vdash a_1 = a_2 \in A$

Equality for application

$\lambda v.b\,(a) \quad \to \quad b\,[a/v]$

Reduction rule for λ

Table 10: Rules for dependent function (part 2)

$H \vdash A_1 \to B_1 = A_2 \to B_2 \in \mathbb{U}_i \qquad \text{ext } \cdot$
BY undependentFunctionEquality
$H \vdash A_1 = A_2 \in \mathbb{U}_i$
$H \vdash B_1 = B_2 \in \mathbb{U}_i$

Equality rule

$H \vdash \mathbb{U}_i \qquad \text{ext } A \to B$
BY undependentFunctionFormation
$H \vdash \mathbb{U}_i \qquad \text{ext } A$
$H \vdash \mathbb{U}_i \qquad \text{ext } B$

Formation rule

$H, f\colon x \to AB, J \vdash T \qquad \text{ext } t\,[f\,(a)\,/y]$
BY undependentFunctionElimination *assumption–index* $\{\$i\}\; y$
$H, f\colon A \to B, J \vdash A \qquad \text{ext } a$
$H, f\colon A \to B, J, y\colon B \vdash T \qquad \text{ext } t$

Elimination rule

Table 11: Rules for undependent functions

3. there are maps π_1 and π_2 such that, for any a where $\alpha(a)$ and b where $\beta_a(b)$, there is an x such that $\varphi(x)$, and $\pi_1(x) = a$ and $\pi_2(x) = b$,

4. there is a map *pair* such that for any x where $\varphi(x)$, there is an a with $\alpha(a)$ and b with $\beta_a(b)$ such that $pair(a, b) = x$.

Typically the map $pair(a, b)$ is denoted by $\langle a, b\rangle$, $\pi_1(x)$ by $x.1$, and $\pi_2(x)$ by $x.2$. The maps have formal types $pair\colon a\colon A \to B \to a\colon A \times B$, $\pi_1\colon a\colon A \times B \to A$, and $\pi_2\colon x\colon a\colon A \times B \to B(x.1)$. In the NuPRL implementation, the π_1 and π_2 projection functions are combined into a single combinator with two binding variables for each of the projections. The combinator is called *spread.* The rules for the product type are listed in Table 12.

In the same manner as for the function type, it is convenient to define an undependent product type. In the propositions-as-types correspondence, the undependent product corresponds to constructive conjunction. The simplified rules are given in Table 13.

$H \vdash x_1{:}\, A_1 \times B_1 = x_1{:}\, A_2 \times B_2 \in \mathbb{U}_i$ ext $\cdot$
BY productEquality y
$H \vdash A_1 = A_2 \in \mathbb{U}_i$
$H, y{:}\, A_1 \vdash B_1\,[y/x_1] = B_2\,[y/x_2] \in \mathbb{U}_i$

Equality rule

$H \vdash \mathbb{U}_i$ ext $x{:}\, A \times B$
BY productFormation a x
$H \vdash A = A \in \mathbb{U}_i$
$H, x{:}\, A \vdash \mathbb{U}_i$ ext B

Formation rule

$H \vdash \langle a_1, b_1 \rangle = \langle a_2, b_2 \rangle \in x{:}\, A \times B$ ext $\cdot$
BY pairEquality *level–exp* $\{i\}$ y
$H \vdash a_1 = a_2 \in A$
$H \vdash b_1 = b_2 \in B\,[a_1/x]$
$H, y{:}\, A \vdash B\,[y/x] = B\,[y/x] \in \mathbb{U}_i$

Member equality rule

$H \vdash x{:}\, A \times B$ ext $\langle a, b \rangle$
BY pairFormation *level–exp* $\{i\}$ a y
$H \vdash a = a \in A$
$H \vdash B\,[a/x]$ ext b
$H, y{:}\, A \vdash B\,[y/x] = B\,[y/x] \in \mathbb{U}_i$

Member formation rule

$H, z{:}\, x{:}\, A \times B, J \vdash T$ ext $\mathbf{let}\ \langle u, v \rangle = z\ \mathbf{in}\ t$
BY productElimination *assumption–index* $\{i\}$ u v
$H, z{:}\, x{:}\, A \times B, u{:}\, A, v{:}\, B\,[u/x], J\,[\langle u, v \rangle/z] \vdash T\,[\langle u, v \rangle/z]$ ext t

Elimination rule

$H \vdash \mathbf{let}\ \langle x_1, y_1 \rangle = e_1\ \mathbf{in}\ b_1 = \mathbf{let}\ \langle x_2, y_2 \rangle = e_2\ \mathbf{in}\ b_2 \in T\,[e_1/z]$ ext $\cdot$
BY spreadEquality z T $w{:}\, A \times B$ u v a
$H \vdash e_1 = e_2 \in w{:}\, A \times B$
$H, u{:}\, A, v{:}\, B\,[u/w], a{:}\, e_1 = \langle u, v \rangle \in w{:}\, A \times B \vdash b_1\,[u, v/x_1, y_1] = b_2\,[u, v/x_2, y_2] \in T\,[\langle u, v \rangle/z]$

Eliminator equality

$\mathbf{let}\ \langle \langle t_1, t_2 \rangle, x_1 \rangle = x_2\ \mathbf{in}\ b \quad \rightarrow \quad b\,[t_1, t_2/x_1, x_2]$

Reduction rule for *spread*

Table 12: Rules for dependent product

Disjoint union The disjoint union type, denoted by $A \mid B$ for two types A and B, is the type whose members are either in A or in B and are tagged as such. This type corresponds to the constructive disjunction. Type term $A \mid B$ is a type with membership φ if-and-only-if:

Uniform representation	
$A \mid B$	*union* $\{\}\,(A; B)$
case e **of** *inl* $x \Rightarrow l \mid$ *inr* $y \Rightarrow r$	*decide* $\{\}\,(x; x.l; y.r)$

1. A is a type with membership α,

2. B is a type with membership β,

$H \vdash A_1 \times B_1 = A_2 \times B_2 \in \mathbb{U}_i$ ext $\cdot$ **BY** undependentProductEquality $H \vdash A_1 = A_2 \in \mathbb{U}_i$ $H \vdash B_1 = B_2 \in \mathbb{U}_i$ Equality rule
$H \vdash \mathbb{U}_i$ ext $A \times B$ **BY** undependentProductFormation $H \vdash \mathbb{U}_i$ ext A $H \vdash \mathbb{U}_i$ ext B Formation rule
$H \vdash \langle a_1, b_1 \rangle = \langle a_2, b_2 \rangle \in A \times B$ ext $\cdot$ **BY** undependentPairEquality $H \vdash a_1 = a_2 \in A$ $H \vdash b_1 = b_2 \in B$ Member equality rule
$H \vdash A \times B$ ext $\langle a, b \rangle$ **BY** pairFormation $H \vdash A$ ext a $H \vdash B$ ext b Member formation rule

Table 13: Rules for undependent product

3. for any element x such that $\varphi(x)$, there is either an element a such that $\alpha(a)$ and $x = \mathit{inl}\, a$, or there is an element b such that $\beta(b)$ and $x = \mathit{inr}\, b$.

We can also provide a decision combinator that can be used on an element to determine the element's tag. The rules for this type are given in Table 14.

A.2.6 Control rules

There are a few proof rules that do not apply to any types in particular. For instance, the standard structural rules are consistent in NuPRL. These special rules are listed in Table 15.

The rules for *axiom*, *thinning*, and *cut* are fairly standard. The *introduction* rule allows the extract of a proof to be provided explicitly. Of special note are the substitution rules. Since the semantics of the sequent require that the goal be pointwise-functional over the hypotheses, it is necessary to provide a type for the domain of substitution, and it is necessary to show that the goal is functional over that type.

A.2.7 Extra NuPRL types

The previous two sections define a basic type theory that is necessary for almost every task. In this section we cover extra types that exists in the system to enhance its expressivity, or to assist with proofs.

Uniform representation	
$\{x{:}\,A \mid B\}$	$\mathit{set}\,\{\}\,(A; x.B)$

"Set" type The "set" type is used for subtyping in NuPRL. The name is really a misnomer, since this type is no more a set than any other type, but the syntax looks very similar to a set construction. The term $\{x{:}\,A \mid B\}$ denotes elements x of type A that also satisfy the property B (which is also a type). Thus, this construction is analogous to the dependent product $x{:}\,A \times B$ where the members "forget" the inhabitant for B. In fact, the semantics we specify require that for any element in the set type, there is a corresponding element in the dependent product type. The term $\{x{:}\,A \mid B\}$ is a type with membership φ if-and-only-if:

$$\begin{array}{ll} H \vdash A_1 \mid B_1 = A_2 \mid B_2 \in \mathbb{U}_i & \text{ext} \cdot \\ \textbf{BY } \text{unionEquality} & \\ H \vdash A_1 = A_2 \in \mathbb{U}_i & \\ H \vdash B_1 = B_2 \in \mathbb{U}_i & \end{array}$$

Equality rule

$$\begin{array}{ll} H \vdash \mathbb{U}_i & \text{ext } A \mid B \\ \textbf{BY } \text{unionFormation} & \\ H \vdash \mathbb{U}_i & \text{ext } A \\ H \vdash \mathbb{U}_i & \text{ext } B \end{array}$$

Formation rule

$$\begin{array}{ll} H \vdash \mathit{inl}\, a_1 = \mathit{inl}\, a_2 \in A \mid B & \text{ext} \cdot \\ \textbf{BY } \text{inlEquality } \mathit{level\text{-}exp}\,\{i\} & \\ H \vdash a_1 = a_2 \in A & \\ H \vdash B = B \in \mathbb{U}_i & \end{array}$$

Member equality for *inl*

$$\begin{array}{ll} H \vdash \mathit{inr}\, b_1 = \mathit{inr}\, b_2 \in A \mid B & \text{ext} \cdot \\ \textbf{BY } \text{inlEquality } \mathit{level\text{-}exp}\,\{i\} & \\ H \vdash b_1 = b_2 \in B & \\ H \vdash A = A \in \mathbb{U}_i & \end{array}$$

Member equality for *inr*

$$\begin{array}{ll} H \vdash A \mid B & \text{ext } \mathit{inl}\ a \\ \textbf{BY } \text{inlFormation } \mathit{level\text{-}exp}\,\{i\} & \\ H \vdash A & \text{ext } a4 \\ H \vdash B = B \in \mathbb{U}_i & \end{array}$$

Member formation rule for *inl*

$$\begin{array}{ll} H \vdash A \mid B & \text{ext } \mathit{inr}\ b \\ \textbf{BY } \text{inlFormation } \mathit{level\text{-}exp}\,\{i\} & \\ H \vdash B & \text{ext } b \\ H \vdash A = A \in \mathbb{U}_i & \end{array}$$

Member formation rule for *inr*

$$\begin{array}{ll} H, z{:}\, A \mid B, J \vdash T & \text{ext } \textbf{case } z \textbf{ of } \mathit{inl}\ x \Rightarrow \mathit{left} \mid \mathit{inr}\ y \Rightarrow \mathit{right} \\ \textbf{BY } \text{unionElimination } \mathit{assumption\text{-}index}\,\{i\}\ x\ y & \\ H, z{:}\, A \mid B, x{:}\, A, J\,[\mathit{inl}\ x/z] \vdash T\,[\mathit{inl}\ x/z] & \text{ext } \mathit{left} \\ H, z{:}\, A \mid B, y{:}\, B, J\,[\mathit{inr}\ y/z] \vdash T\,[\mathit{inr}\ y/z] & \text{ext } \mathit{right} \end{array}$$

Elimination rule

$$\begin{array}{ll} H \vdash \textbf{case } e_1 \textbf{ of } \mathit{inl}\, x_1 \Rightarrow l_1 \mid \mathit{inr}\, y_1 \Rightarrow r_1 = \textbf{case } e_2 \textbf{ of } \mathit{inl}\, x_2 \Rightarrow l_2 \mid \mathit{inr}\, y_2 \Rightarrow r_2 \in T\,[e_1/z] & \text{ext} \cdot \\ \textbf{BY } \text{decideEquality } z\ T\ A \mid B\ u\ v\ w & \\ H \vdash e_1 = e_2 \in A \mid B & \\ H, u{:}\, A, w{:}\, e_1 = \mathit{inl}\, u \in A \mid B \vdash l_1\,[u/x_1] = l_2\,[u/x_2] \in T\,[\mathit{inl}\, u/z] & \\ H, v{:}\, B, w{:}\, e_1 = \mathit{inr}\ v \in A \mid B \vdash r_1\,[v/y_1] = r_2\,[v/y_2] \in T\,[\mathit{inr}\ v/z] & \end{array}$$

Eliminator rule

$$\begin{array}{lcl} \textbf{case } \mathit{inl}\, t \textbf{ of } \mathit{inl}\, x_1 \Rightarrow b_1 \mid \mathit{inr}\, x_2 \Rightarrow b_2 & \rightarrow & b_1\,[t/x_1] \\ \textbf{case } \mathit{inr}\, t \textbf{ of } \mathit{inl}\, x_1 \Rightarrow b_1 \mid \mathit{inr}\, x_2 \Rightarrow b_2 & \rightarrow & b_2\,[t/x_2] \end{array}$$

Reduction rule for *decide*

Table 14: Rules for disjoint union

1. A is a type with membership α,
2. for each a where $\alpha(a)$, $B(a)$ is a type with membership β_a,
3. for any element x, if $\varphi(x)$ then $\alpha(x)$, and there exists a y such that $\beta_x(y)$.

Thus, every element $y \in \{x{:}\, A \mid B\}$ is also an element of A, and $B(y)$ is inhabited.

Since the inhabitant for B is not included in the members of the set type, one might wonder how this type could be useful in a constructive type theory. However, there are a few situations where the proof of B is not necessary:

- If the proof for the current sequent is computationally trivial (as it is for all equality goals), the computational content of B will not used used for the current proof, and B can be exposed.
- If the inhabitant for B can be constructed, then it can be exposed. This would the case if B is computationally trivial (if it is an equality).

$H, x\colon A, J \vdash A \quad$ ext x
BY hypothesis *assumption–index* $\{i\}$

Hypothesis

$H, x\colon A, J \vdash T \quad$ ext t
BY thin *assumption–index* $\{i\}$
$H, J \vdash T \quad$ ext t

Thinning

$H, J \vdash T \quad$ ext $t\,[s/x]$
BY cut *assumption–index* $\{i\}$ S x
$H, J \vdash S \quad$ ext s
$H, x\colon S, J \vdash T \quad$ ext t

Cut

$H \vdash T \quad$ ext t
BY introduction t
$H \vdash t = t \in T$

Introduction

$H \vdash T_1\,[t_1/x] \quad$ ext t
BY substitution *level–exp* $\{i\}$ $(t_1 = t_2 \in T_2)$ x T_1
$H \vdash t_1 = t_2 \in T_2$
$H \vdash T_1\,[t_2/x] \quad$ ext t
$H, x\colon T_2 \vdash T_1 = T_1 \in \mathbb{U}_i$

Substitution

$H, x\colon A, J \vdash T \quad$ ext t
BY hypothesisReplacement *assumption–index* $\{j\}$ B *level–exp* $\{i\}$
$H, x\colon B, J \vdash T \quad$ ext t
$H, x\colon A, J \vdash A = B \in \mathbb{U}_i$

Hypothesis replacement

Table 15: Structural rules

When a hypothesis that belongs to a set type is eliminated, the proof of B is listed as a "hidden" hypothesis. Hidden hypotheses can't be manipulated in any way, but the system will automatically unhide them if they are among the "standard" group of computationally trivial types or if the goal is. The notion of computationally trivial is defined syntactically, and includes the equality and $<$ types. This notion could be sufficiently generalized to employ the full power of the proof theory, but little would be gained. The rules for the set type are listed in Table 16.

Uniform representation	
$\bigcap_{x \in A} B$	*intersection* $\{\}\,(A; x.B)$

Intersection type The intersection type $\bigcap_{x \in A} B$ is the intersection of all types $B(a)$ over all elements $a \in A$. If A is empty, then the intersection is equivalent to the *Void* type. The intersection type is useful for specifying polymorphism. For instance, it is possible to prove that $\lambda x.x$ has type $T \to T$ for any type T. This can be stated as the proposition $\forall T\colon \mathbb{U}_i.\lambda x.x \in T \to T$, or it can be stated using the intersection type as $\bigcap_{\lambda x.x \in \bigcap_{T \in_i}} T$.

The intersection type has similarities with the function type. An element of the intersection belongs to each type in the intersection. We can infer that the element belongs to any type in the intersection, and the elimination rule essentially corresponds to a function application. The rules for the intersection type are listed in Table 17.

$H \vdash \{x_1 : A_1 \mid B_1\} = \{x_2 : A_2 \mid B_2\} \in \mathbb{U}_i$ ext $\cdot$
BY setEquality x
$H \vdash A_1 = A_2 \in \mathbb{U}_i$
$H, x : A_1 \vdash B_1[x/x_1] = B_2[x/x_2] \in \mathbb{U}_i$

Equality rule

$H \vdash \mathbb{U}_i$ ext $\{x : A \mid B\}$
BY setFormation A x
$H \vdash A = A \in \mathbb{U}_i$
$H, y : A \vdash \mathbb{U}_i$ ext B

Formation rule

$H \vdash a_1 = a_1 \in \{x : A \mid B\}$ ext $\cdot$
BY setMemberEquality
$H \vdash a_1 = a_2 \in A$
$H \vdash B[a_1/x]$
$H, y : A \vdash B[y/x] = B[y/x] \in \mathbb{U}_i$

Member equality rule

$H \vdash \{x : A \mid B\}$ ext a
BY setMemberFormation *level–exp* $\{i\}$ a y
$H \vdash a = a \in A$
$H \vdash B[a/x]$
$H, y : A \vdash B[y/x] = B[y/x] \in \mathbb{U}_i$

Member formation rule

$H, u : \{x : A \mid B\}, J \vdash T$ ext $t[y/u]$
BY setElimination *assumption–index* $\{i\}$ y v
$H, u : \{x : A \mid B\}, [y] : A, v : B[y/x], J[y/x] \vdash T[y/u]$ ext t

Elimination rule

Table 16: Rules for set type

Uniform representation	
$a \quad b$	$sqequal\,\{\}\,(a; b)$

Term ("squiggle") equality The standard substitution rule is a powerful proof tool, but it also has drawbacks. For each substitution that is performed, the rule requires that the goal be proved functional. This is a stronger condition than is necessary, since all that is really required is that the goal be functional over the substitution. This can be generalized to a proof-theoretic notion of term congruence. The terms can be partitioned into equivalence classes over which all terms are congruent. For instance, in the type theory just given, no context can distinguish the terms 2 and $1 + 1$. In order to do so, we would have to add a non-functional term over arithmetic expressions.

Given the proof system, we can proof-theoretically determine the equivalence classes for the congruence. We denote the congruence of two terms t_1 and t_2 by the type $t_1 \quad t_2$ (hence the name "squiggle" equality). In the proof system given so far for NuPRL there are two main classes for the congruence:

- Equal terms in *base* types are congruent.
- Terms that are computationally equal are congruent.

It may be possible to define coarser equivalence classes, but these cover most of the practical cases. We cover the case of equal arithmetic expressions, as well as terms that compute to a syntactically equal term.

If $t_1 \quad t_2$, then it is consistent to replace an occurrence of t_1 with an occurrence of t_2 in any context. This provides a stronger substitution where it is not necessary to prove functionality of the context over the substitution. Since squiggle equality is a proof theoretic concept dealing with

$H \vdash \bigcap_{x_1 \in A_1} B_1 = \bigcap_{x_2 \in A_2} B_2 \in \mathbb{U}_i$ ext $\cdot$
BY intersectionEquality y
$H \vdash A_1 = A_2 \in \mathbb{U}_i$
$H, y\colon A_1 \vdash B_1\,[y/x_1] = B_2\,[y/x_2] \in \mathbb{U}_i$

Equality rule

$H \vdash \mathbb{U}_i$ ext $\bigcap_{x \in A} B$
BY intersectionFormation x A
$H \vdash A = A \in \mathbb{U}_i$
$H, x\colon A \vdash \mathbb{U}_i$ ext B

Formation rule

$H \vdash b_1 = b_2 \in \bigcap_{x \in A} B$ ext $\cdot$
BY intersectionMemberEquality *level–exp* $\{i\}$ z
$H, z\colon A \vdash b_1 = b_2 \in B\,[z/x]$
$H \vdash A = A \in \mathbb{U}_i$

Member equality

$H \vdash b_1 = b_2 \in B\,[a/x]$ ext $\cdot$
BY intersectionMemberCaseEquality $\bigcap_{x \in A} B$ a
$H \vdash f_1 = f_2 \in \bigcap_{x \in A} B$
$H \vdash a_1 = a_2 \in A$

Member equality on a case

$H \vdash \bigcap_{x \in A} B$ ext b
BY intersectionMemberFormation *level–exp* $\{i\}$ z
$H, z\colon A \vdash B\,[z/x]$ ext b
$H \vdash A = A \in \mathbb{U}_i$

Member formation

$H, f\colon \bigcap_{x \in A} B, J \vdash T$ ext $t\,[f, \cdot/y, v]$
BY intersectionElimination *assumption–index* $\{i\}$ a y v
$H, f\colon \bigcap_{x \in A} B, J \vdash a = a \in A$
$H, f\colon \bigcap_{x \in A} B, J, y\colon B\,[a/x], v\colon v = f \in B\,[a/x] \vdash T$ ext t

Elimination rule

Table 17: Rules for intersection

the syntax of terms, it is highly non-functional. It many cases it is not possible to prove that term term a b is a type, even if the term itself is provable. The rule for equality is based on a decision procedure, where the term a b is a type if all subterms of a and b that differ are base type, or combinations thereof. The rules for squiggle equality are given in Table 18.

A.2.8 Recursive type in NuPRL

Uniform representation	
$\mu\,(A, x.B; a)$	*prec* $\{\}\,(A, x.B; a)$
precind $(a; p, h.g)$	*precind* $\{\}\,(a; p, h.g)$

The last builtin NuPRL types we will discuss are the recursive types. These types are discussed more thoroughly in Mendler's thesis [13]. The term expressing a recursive type construction is $\mu\,(A, x.B; a)$, where X and a are bound in T. This construction is called a *parameterized recursive type*, and the type is defined by the term B, where A represents the least fixed point of the type being defined, x represents a parameter to the type definition, and a is the initial value of the parameter. When the parameter

$H \vdash a \quad b = c \quad d \in \mathbb{U}_i$ ext $\cdot$
BY sqequalEquality (decision-procedure)

Equality

$H \vdash \cdot = \cdot \in a \quad b$ ext $\cdot$
BY sqequalMemberEquality
$H \vdash a \quad b$

Member equality

$H, x{:}\, a \quad b, J \vdash T$ ext t
BY sqequalElimination *assumption–index* $\{i\}$
$H, x{:}\, a \quad b, J\,[\cdot/x] \vdash T\,[\cdot/x]$ ext t

Elimination rule

$H \vdash a \quad b$ ext $\cdot$
BY sqequalInt
$H \vdash a = b \in \mathbb{Z}$

$H \vdash a \quad b$ ext $\cdot$
BY sqequalAtom
$H \vdash a = b \in Atom$

$H \vdash a \quad b$ ext $\cdot$
BY sqequalEqual *level–exp* $\{i\}$ $(c = d \in T)$
$H \vdash a = b \in (c = d \in T)$

Rules for base types

$H \vdash a \quad a$ ext $\cdot$
BY sequalRelfexivity

Relfelxivity

$H \vdash T\,[a/x]$ ext t
BY sqequalSubstitition $(a \quad b)$ x T
$H \vdash a \quad b$
$H \vdash T\,[b/x]$

Goal substitution

$H, y{:}\, S\,[a/x]\,, J \vdash T$ ext t
BY sqequalHypothesisSubstition *assumption–index* $\{i\}$ $(a \quad b)$ x T
$H, y{:}\, S\,[a/x]\,, J \vdash a \quad b$
$H, y{:}\, S\,[b/x]\,, J \vdash T$ ext t

Hypothesis substition

Table 18: Squiggle equality rules

is not used, the construction is represented by the term $\mu\,(A.B)$.

A simple example is a recursive type defining lists of integers. Any type with a know inhabitant can be used for the tail; for convenience we use *Unit*. The type is defined as:

$$\mu\,(T.\mathit{Unit} \mid \mathbb{Z} \times T) \quad \textit{(list of integers)}$$

The type constructor $\mathit{Unit} \mid \mathbb{Z} \times T$ is monotonic in inclusion on T (that is, if $T_1 \subseteq T_2$, then $\mathit{Unit} \mid \mathbb{Z} \times T_1 \subseteq \mathit{Unit} \mid \mathbb{Z} \times T_2$), so a fixed point does exist. For this coding of lists, the familiar constructors are:

Constructor	Coding
[]	$inl\,\cdot$
a::b	$inr\,\langle a, b\rangle$

We can also define trees with this recursive construction. For instance, finite branching trees of $\mathbb{Z}$ can be defined with the construction

$$\mu\left(T.\,Unit \mid v{:}\,\mathbb{Z} \times i{:}\,\mathbb{N} \times T^i\right).$$

The leaves of the tree are $inl\,\cdot$, and the internal nodes each have an integer value v, a natural number i this is the number of children of the node, and the i subtrees, where $T^i \equiv \underbrace{T \times T \times \cdots \times T}_{i \text{ times}} \times Unit$ is computed using the induction combinator for $\mathbb{Z}$.

The parameterized type constructor becomes handy if we want to specify dependencies in the type being constructed. For instance, the construction

$$\mu\,(T, n.\,Unit \mid v{:}\,\{\ldots(n-1)\}\,i{:}\,\mathbb{N} \times (T\,(v))\times; m)$$

defines a finite branching tree of integers where the children always have smaller values than their parents, and the root has value smaller than m. Using a slightly more complicated construction where the parameter ranges over $\mathbb{Z} \cup \{\infty\}$, the restriction on the root can be removed.

This recursive type can be used to define W-types. The W-types can be used to define trees by describing the nodes without explicitly mentioning the recursion. The basic form for the W-type is $W\,(x.A; B)$, where x is bound in B, A is the type of "labels" of the nodes in the tree, and $B[x]$ is the domain of subtrees of the node with label x. If the label is not used in the subtree domain, the notation is simplified to $W\,(A; B)$. For instance, binary trees of integers can be described using the construction $W\,(\mathbb{Z}; \mathbb{B})$. Each node is labeled by an integer, and the domain of subtrees can be expressed by $\mathbb{B}$, since there are exactly two subtrees. For finite branching trees of integers, we can use the type $W\,(x.\mathbb{Z} \times \mathbb{N}; \{0 \ldots x.2\})$. In this case, each node is labeled with a pair of numbers. The first is an integer that is the standard label, and the second is a natural number that specifies how many children the node has. The domain of the subtrees can then be specified by the integer subrange $\{0 \ldots x.2\}$.

The rules for the parameterized recursive type are given in Table 19. For each well-formed types of the form $\mu\,(T, x.U; v)$ the parameters must all belong to some type A, and for any type function $T{:}\,A \to \mathbb{U}_i$ and parameter $x{:}\,A$, the term U must define a type in $\mathbb{U}_i$. In addition, the type definition U must be monotonic in T. That is, if P_1 and P_2 are two type valued functions on A, and the range of P_2 is smaller than P_1, then $U[P_2, a] \subseteq U[P_1, a]$ for any argument $a \in A$. These restrictions define a class of recursive types with fixed points, that are semantically meaningful in NuPRL.

The major restriction on the well-formed types is that the type defined by U be a well-formed type for *any* type valued function $T{:}\,A \to \mathbb{U}_i$. In the next section, we will cover this restriction in more detail. This is a major dilemma, for if we assume any structure on the type returned by the function T, then we are also assuming that the type being defined is well-formed. This would be a cyclic argument. Note also, that the definition is impredicative.

There is no formation rule for the recursive type because any incremental construction would be recursive, requiring an argument by induction that would not mesh well with the proof refinement process.

A.3 Computation

The final issue we will discuss in this section is the issue of computation, or *direct computation* as it is called in NuPRL. The computation rules expressed for each type are of the form $t \to t'$, where t and t' are terms containing free variables. These computation rules can be used to express equivalences of terms. For example in any context $\lambda v.v+1\,(2) \equiv 4$. The computational equivalences hold even of the terms do not have a well-defined type: for instance $(\lambda v.\text{“}hello'' + v)\,(\text{“}world'') \equiv \text{“}hello'' + \text{“}world''$, even though the terms themselves have no meaning. One of the issues we will be working towards is termination—is a term equivalent to an irreducible term? In the process we discuss

- *substitution instances* of rules, which are all the applicable instances of rules,

$H \vdash \mu(A_1, x_1.B_1; a_1) = \mu(A_2, x_2.B_2; a_2) \in \mathbb{U}_i$ ext $\cdot$
BY precEquality $A\ x\ y\ z\ T\ P_1\ P_2$
$H \vdash a_1 = a_2 \in A$
$H, x{:}\, A, T{:}\, A \to \mathbb{U}_i \vdash B_1\,[T, x/T_1, x_1] = B_2\,[T, x/T_2, x_2] \in \mathbb{U}_i$

$$\left[\begin{array}{l} H, P_1{:}\, A \to \mathbb{U}_i, P_2{:}\, A \to \mathbb{U}_i, \\ z{:}\, x{:}\, A \to y{:}\, P_1(x) \to y \in P_2(x), \\ x{:}\, A, y{:}\, B_1\,[P_1, x/T_1, x_1] \end{array}\right] \vdash y = y \in P_2, xT_1, x_1$$

Equality rule

$H \vdash a_1 = a_2 \in \mu(T, x.B; a)$ ext $\cdot$
BY precMemberEquality *level–exp* $\{i\}$
$H \vdash \mu(T, x.B; a) = \mu(T, x.B; a) \in \mathbb{U}_i$
$H \vdash a_1 = a_2 \in B\,[\lambda a.\mu(T, x.B; a), a/T, x]$

Member equality rule

$H \vdash \mu(T, x.B; a)$ ext t
BY precMemberFormation *level–exp* $\{i\}$
$H \vdash B\,[\lambda a.\mu(T, x.B; a), a/T, x]$ ext t
$H \vdash \mu(T, x.B; a) = \mu(T, x.B; a) \in \mathbb{U}_i$

Member formation rule

Table 19: Rules for the parameterized recursive type (part 1)

$H, r{:}\, \mu(T, x.B; a), J \vdash G\,[a/p]$ ext $precind\,(a; p, h.g)$
BY precElimination $p\ G$ *assumption–index* $\{n\}$ *level–exp* $\{i\}$ $A\ Z\ u\ h\ y$
$H, r{:}\, \mu(T, x.B; a), J \vdash a = a \in A$

$$\left[\begin{array}{l} H, r{:}\, \mu(T, x.B; a), J, \\ Z{:}\, A \to \mathbb{U}_i, \\ u{:}\, p{:}\, a{:}\, A \times Z(a) \to p \in a{:}\, A \times \mu(T, x.B; a), \\ h{:}\, p{:}\, a{:}\, A \times Z(a) \to G, p{:}\, a{:}\, A \times B\,[Z, a/T, x] \end{array}\right] \vdash G \quad \text{ext } g$$

Rule for elimination

$H \vdash precind\,(r_1; h_1, z_1.t_1) = precind\,(r_2; h_2, z_2.t_2) \in S\,[r_1/x]$ ext $\cdot$
BY precIndEquality $x\ S\ (a{:}\, A \times \mu(T, y.B; a))$ *level–exp* $\{i\}$ $Z\ u\ h\ z$
$H \vdash r_1 = r_2 \in a{:}\, A \times \mu(T, y.B; a)$

$$\left[\begin{array}{l} H, Z{:}\, A \to \mathbb{U}_i, \\ u{:}\, z{:}\, a{:}\, A \times Z(a) \to z \in a{:}\, A \times \mu(T, x.B; a), \\ h{:}\, z{:}\, a{:}\, A \times Z(a) \to S\,[z/x], z{:}\, a{:}\, A \times B\,[Z, a/T, y] \end{array}\right] \vdash t_1\,[h, z/h_1, z_1] = t_2\,[h, z/h_2, t_2] \in S\,[z/x]$$

Equality rule in induction combinator

$H, z{:}\, \mu(T, x.B; a), J \vdash G$ ext $g\,[z/y]$
BY precUnrollElimination *assumption–index* $\{i\}$ $y\ u$
$H, z{:}\, \mu(T, x.B; a), J, y{:}\, B\,[\lambda a.\mu(T, x.B; a), a/T, x], u{:}\, x = y \in B\,[\lambda a.\mu(T, x.B; a) / \vdash]\, G\,[y/z]$ ext g

Unrolling

$$precind\,(a; p, h.g) \quad \to \quad g\,[\lambda a.\, precind\,(a; p, h.g), a/p, h]$$

Reduction rule for *precind*

Table 20: Rules for the parameterized recursive type (part 2)

- *contexts*, which are subterms that direct computation can be applied to,
- *evaluation*, which reduces a term in an equivalent irreducible term,

- and *termination*, which specifies which terms are equivalent to irreducible terms.

To begin, we would like to specify the meaning of a computation rule. Intuitively, a computation rule such as $(\lambda v.b)\,(a) \rightarrow b\,[a/v]$ specifies a set of *actual* computation rules. In this example, v stands for a variable, and b and a stand for arbitrary terms. The variables v, b, and a are *second order* variables because they stand for other terms. The actual rules are those obtained when the second order variables are *instantiated* with appropriate terms. We can precisely define this concept with *substitution instances*, which are defined in the following few paragraphs. In this section, terms are considered equivalent up to α-equality.

Definition 1 *Let t be a term, let $\overline{x} \equiv x_1, x_2, \ldots, x_n$ be a subset of the free variables of t, and let $\overline{s} \equiv s_1, s_2, \ldots, s_n$ be a vector of terms. Then $t^* \equiv t\,[\overline{s}/\overline{x}]$ is a substitution instance of t.*

The same concept of substitution can be extended to computation rules.

Definition 2 *Let $r \equiv t_1 \rightarrow t_2$ be a computation rule, let $\overline{x} \equiv x_1, x_2, \ldots, x_n$ be a subset of the free variables of t_1 and t_2, and let $\overline{s} \equiv s_1, s_2, \ldots, s_n$ be a vector of terms. then $r^* \equiv t\,[\overline{s}/\overline{x}] \rightarrow t'\,[\overline{s}/\overline{x}]$ is a substitution instance of r.*

The terms of a computation rule are defined as follows:

Definition 3 *Let $t \rightarrow t'$ be a computation rule. A substitution instance of t is called a* redex*, and the corresponding substitution instance of t' is called the* contractum *of t. A term that is not a redex is a* value*.*

The actual reductions are the substitution instances of the computation rules. In NuPRL it is consistent to apply a reduction to a term in any context. If two terms t and t' are equivalent by $t \rightarrow t'$, then they are equivalent in any context.[2] But what is a *context*? Informally, a context is a term with a *hole* in it, where the hole is a subterm. If we specify a context by [], then [] is a context, $\lambda v.[]$ is a context, but $\lambda[].b$ is not. In NuPRL, valid contexts can be easily specified using uniform representations.

Definition 4 *Any term in the uniform representation*

$$C \equiv op\,\{params\}\,(\overline{x_1}.a_1; \ldots, \overline{x_{i-1}}.a_{i-1}; \overline{x_i}.[]; \overline{x_{i+1}}.a_{i+1}; \ldots; \overline{x_n}.a_n)$$

is a context*, where*

$$C[t] \equiv op\,\{params\}\,(\overline{x_1}.a_1; \ldots, \overline{x_{i-1}}.a_{i-1}; \overline{x_i}.t; \overline{x_{i+1}}.a_{i+1}; \ldots; \overline{x_n}.a_n)$$

Using contexts, we can extend the reduction relation to an equivalence on terms. First we define the compatible closure (see Barendregt [4], pages 50–51) of the reduction relation. The relation $\twoheadrightarrow$ is the compatible closure of the reduction relation:

1. if $t \rightarrow t'$, then $t \twoheadrightarrow t'$,
2. if $t \rightarrow t'$, then $C[][t] \twoheadrightarrow C[][t']$.

The relation $\twoheadrightarrow_*$ is the reflexive, transitive closure of $\twoheadrightarrow$:

1. if $t \twoheadrightarrow t'$, then $t \twoheadrightarrow_* t'$,
2. $t \twoheadrightarrow t$,
3. if $t \twoheadrightarrow_* t'$ and $t' \twoheadrightarrow_* t''$, then $t \twoheadrightarrow_* t''$.

The equivalence relation $=_\beta$ is defined as follows:

1. if $t \twoheadrightarrow_* t'$, then $t =_\beta t'$,

[2] This is really due to the *proof theory*; it is possible to imagine an extension to the NuPRL theory where this would not be true. A classic example would be a type theory that accounts for computational complexity. Thus, although $f\,(a) \equiv f'\,(a)$, the two functions f and f' may have wildly different computational complexity. To say they are equivalent by direct computation would invalidate all claims on their complexity. The standard NuPRL type theory equates terms that can be reduced to a common term, because it is modeled on pure mathematical content. The two terms t and t' *stand* for the same mathematical concept (which may be nonsense).

2. if $t =_\beta t'$, then $t' =_\beta t$,
3. if $t =_\beta t'$ and $t' =_\beta t''$, then $t =_\beta t''$.

As stated before, in NuPRL if $t =_\beta t'$ then t and t' are interchangeable. Put more concretely, if $t_i =_\beta t'_i$ then the two sequents

$$x_1{:}\, t_1, \ldots, x_i{:}\, t_i, \ldots, x_n{:}\, t_n \vdash T$$

and

$$x_1{:}\, t_1, \ldots, x_i{:}\, t'_i, \ldots, x_n{:}\, t_n \vdash T$$

have equivalent meaning. Similarly, if $T =_\beta T'$, the sequents

$$x_1{:}\, t_1, \ldots, x_n{:}\, t_n \vdash T$$

and

$$x_1{:}\, t_1, \ldots, x_n{:}\, t_n \vdash T'$$

are equivalent.

There is also the notion of a canonical evaluation of terms. If the canonical evaluator terminates, it produces a value equivalent to its argument. The evaluation is *leftmost–outermost*; it is a *lazy* evaluator. Of course, other evaluation orders are possible, and they may be preferred in some contexts. All evaluators are functional: if they halt, they provide equal values on equal arguments.

The canonical evaluation relation $\downarrow$ is defined as follows:

1. if t is a value, then $t \downarrow t$,
2. if $t \rightarrow t'$ and $t' \downarrow t''$, then $t \downarrow t''$,
3. let $s \equiv \mathit{opname}\, \{\overline{p}\}\, (\overline{x_1}.t_1; \cdots; \overline{x_n}.t_n)$ and let i be the smallest integer in the range $1 \leq i \leq n$ where
   (a) there are terms $t'_1, \ldots, t'_i$ such that $t_1 \downarrow t'_1, \ldots, t_i \downarrow t'_i$,
   (b) there is a computation rule $\mathit{opname}\, \{\overline{p}\}\, (\overline{x_1}.t'_1; \cdots; \overline{x_i}.t'_i; \overline{x_{i+1}}.t_{i+1}; \cdots; \overline{x_n}.t_n) \rightarrow s'$,
   (c) there is an evaluation $s' \downarrow s''$,

   then $s \downarrow s''$.

It is difficult, but possible, to show that given the current computation rules, if any evaluator returns a value, then so does the canonical evaluator. If the canonical evaluator halts, we say that evaluation of a term *terminates*, or more simply that the term terminates.

The fact that a term can be typed is independent of whether the term terminates. The simplest example is the type $\bigcap_{x \in \mathit{Void}} T$, which is a well-formed type for any term T. Every term in NuPRL belongs to this type, because under the assumption that *Void* is inhabited anything is true. Even the term $\Omega \equiv (\lambda x.xx)\,(\lambda x.xx) \in \bigcap_{x \in \mathit{Void}} T$, although $\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \cdots$. However, there are types where membership implies termination—in fact, almost all types in NuPRL require that their members have values. If $i \in \mathbb{Z}$, then i is equivalent to some value that is an integer. The values in $\mathbb{Z}$ are just numbers, and the canonical evaluator can be used to compute the number for i. Similarly if $t \in \mathbb{U}_i$, then t must be equivalent to some value that is a type. The value for t must be some base type, or type construction.