Formal Objects in Type Theory Using Very Dependent Types

Jason J. Hickey*
Department of Computer Science,
Cornell University

Abstract

In this paper we present an extension to basic type theory to allow a uniform construction of abstract data types (ADTs) having many of the properties of objects, including abstraction, subtyping, and inheritance. The extension relies on allowing type dependencies for function types to range over a well-founded domain. Using the propositions–as–types correspondence, abstract data types can be identified with logical theories, and proofs of the theories are the objects that inhabit the corresponding ADT.

1 Introduction

In the past decade, there has been considerable progress in developing formal account of a theory of objects. One property of object oriented languages that make them popular is that they attack the problem of scale: all object oriented languages provide mechanisms for providing software modularity and reuse. In addition, the mechanisms are intuitive enough to be followed easily by novice programmers.

During the same decade, the body of formal mathematics has been steadily increasing, and proof development systems have also been attacking the problem of scale. Algebraic specification techniques are a basis for many approaches, including Breu [5] who uses them to develop an object calculus for software systems. Jackson [17] has developed these techniques in the Nuprl [12] system to implement a considerable amount of constructive abstract algebra.

Subtyping is key to modeling inheritance and reuse, and Aspinali and Compagnoni [4], and Sannella *et. al.* [26] have analyzed subtyping in the presence of type dependencies in order to model formal programming *in-the-large*. Jackson [17] has developed subtyping in Nuprl for use in his account of constructive algebra. Reus and Streicher [24] have translated the laws of module algebra into laws of "deliverables" that can be proved correct in type theory, which they have performed using the *Extended Calculus of Constructions* on the LEGO [18] system. Another line of research has been followed to provide an object-oriented calculus, for instance in Mitchell [22] and Abadi and Cardelli [1]. Record calculi with subtyping are key features of these systems.

In this paper we present an extension to constructive type theory to allow a uniform construction of abstract data types (ADTs) having many of the properties of objects, including abstraction, subtyping, and inheritance. Our work relies on a new type constructor

*Support for this research was provided by the Office of Naval Research through grant N00014-92-J-1764, from the National Science Foundation through grant CCR-9244739

called *the very dependent function type*, $\{f \mid x{:}\, A \rightarrow B\}$. Dependent function types are used in the type theory to specify relationships between the input value of a function and the corresponding output. The very-dependent types extend the dependencies by allowing the specification for the output to depend on more than one input value (in some pre-defined manner).

We have implemented an object calculus using the very-dependent function types in the Nuprl system. Although we use the Nuprl type theory (based on Martin-Löf's [19, 20] type theory) and the Nuprl terminology in this paper, our results should carry over to other proof systems that use dependent types, including *Coq* [11], LEGO [18], *Alf* [9], and PVS [10].

We present the following results:

- a dependent record calculus, forming the foundation for a theory of objects,
- a generalization of the type theory's function type to allow dependencies over a well-founded domain,
- rules and semantics for the new function type,
- a correspondence between abstract data types and theories, and between objects and proofs.

There are several reasons to develop an account of objects within the type theory. The type theory already has several semantics, including set theoretic [16], PER models [2, 21], denotational models [25], and others. By developing our account within the type theory, we provide a solid mathematical foundation for objects. In addition, we can draw on mathematical techniques, especially those of abstract algebra, to contribute toward the understanding and development of object-oriented languages. Finally, we can feed the results back into the type theory to provide a foundation for formal modules and object-oriented programming.

2 Overview

We develop our account of objects within the context of an object calculus in which the only primitives are objects, method selection, and method override. The type theory that we are using is a small, but very expressive fragment of the Nuprl type theory including subtypes, intersections, dependent function types, as well as the terms of the untyped lambda calculus, numbers, and strings. This fragment is shown in more detail in Appendix A. In the type theory, programs are *untyped*, and types are used to classify the untyped programs. This untyped interpretation has a different flavor than the accounts based on variants of System F, and it provides greater flexibility in formalizing the primitives of object calculus. Of course, this greater flexibility also means that type inference is undecidable. We will not discuss the type inference problem in this mathematical account. However, once our account is developed, it is possible to restate the account with explicit type annotations.

The three primitives are expressed as follows. An object $o = [l_i = \lambda x_i.b_i]^{i \in \{1 \ldots n\}}$ is a collection of methods $\lambda x_1.b_1, \ldots, \lambda x_n.b_n$ where the methods have names $l_1, \ldots, l_n$. The methods *bodies* are represented by $b_1, \ldots, b_n$, and the variables $x_1, \ldots, x_n$ represent *self* (in this case the object o). *Method selection*, written $o.l_j$, substitutes the object o for x_j on body b_j. *Method override*, written $o.l_j \Leftarrow \lambda x.b$, creates a new object from o, replacing the method $\lambda x_j.b_j$ with the new method $\lambda x.b$. This operational semantics is summarized in the following table:

For an object $o = [l_i = \lambda x_i.b_i]^{\ i \in \{1 \ldots n\}}$:		
$o.l_j$	$\rightarrow$	$b_j[o/x_j]$
$o.l_j \Leftarrow \lambda x.b$	$\rightarrow$	$[l_j = \lambda x.b, l_i = \lambda x_i.b_i\ ^{i \in \{1 \ldots n\} - \{j\}}]$

Object types are used to classify these untyped objects. Before object types can be developed, the concept of "Self" needs to be explored. In much of the current work on objects, for example Mitchell, Honsell, and Fisher [22], Abadi and Cardelli [1], Amadio and Cardelli [3], Harper and Pierce [13], and Bruce [7], the self-application of method selection is handled directly, and the interpretation often use a recursive type to represent the type for "Self."

To illustrate this point, we use the standard example of a one-dimensional movable point having two methods: the *getX* method retrieves the position of the point (an integer), and the *setX* method takes an integer and produces a point at that location. Using recursive types, the type for the point might be described as follows:

$$\mathit{Point} = \mu(X.[\mathit{getX}\colon \mathbb{Z}; \mathit{setX}\colon \mathbb{Z} \rightarrow X])$$

Although an account of these calculi in the type theory is reasonable (without the use of recursive types [14]), the interaction of self-application and method override with the predicativity of the type theory can be complex enough to obscure the interpretation.

Instead, we adopt the existential interpretation of Pierce and Turner [23] (this is the "state-application semantics" of Abadi and Cardelli [1]), in which each object has a state, and updates are allowed only to the state. This formalization has a simple, predicative, type theoretic interpretation, at the expense of a more complicated translation of object primitives and their types (and the lack of proper method override), and it is more in keeping with the goal of a formal object system for the type theory itself.

Returning to the example, points are modeled in the existential interpretation by including a representation type for state of the point, and wrapping the methods in a subrecord:

$$\begin{array}{rcl} \mathit{Point} & \equiv & \exists \mathit{Rep}\colon \mathbf{Type}.\{\ \mathit{state}\colon \mathit{Rep}; \\ & & \qquad\qquad\quad \mathit{methods}\colon \{\ \mathit{getX}\colon \mathit{Rep} \rightarrow \mathbb{Z}; \\ & & \qquad\qquad\qquad\qquad\quad\ \ \mathit{setX}\colon \mathit{Rep} \rightarrow \mathbb{Z} \rightarrow \mathit{Rep}\}\} \end{array} \tag{1}$$

The $\exists \mathit{Rep}\colon \mathbf{Type}$ is a quantification over all possible representation types. The specific type is hidden by an abstraction barrier, although the specific type is "known" when the object is constructed. Information about specific points is kept within their state, which may have different representations for different instances of points. In general the methods of an object know only about the representation of the object they belong to. Externally, the only interface to the state is through the methods.

Our task in this existential interpretation is to encode the primitives in the type theory in a way that captures the behavior of objects, including the properties of subtyping and method subsumption. The translations of the primitives is rather complex, and we will develop it in the terms of record calculi. As we will show subtyping and subsumption follows from the primitives of records.

2.1 Record encoding

We represent objects with records, where the record $\{l_1 = x_1, \ldots, l_n = x_n\}$ has fields $l_1, \ldots, l_n$ with values $x_1, \ldots, x_n$, respectively. Records provide selection (without self-application) written $r \cdot l$ for a record r on field l, and functional update, written $r \cdot l := x$, which creates a new record from r with the value of field l replaced by x. We further encode records as functions on labels. For simplicity, let the collection of all labels have the type **Atom**, and assume records have as default value the type **Top** on uninteresting elements. The type **Top** contains a single element, which is the equivalence class containing all terms (including the term **Top**).

$$\begin{array}{rcl} \{l_i = x_i{}^{i \in \{1 \ldots n\}}\} & \equiv & \lambda l.\, \mathbf{case}\, l \;\mathbf{of}\; l_1 \colon x_1 | \cdots | l_n \colon x_n | \_\colon \mathbf{Top} \\ r \cdot l_j & \equiv & r(l_j) \\ r \cdot l_j := x & \equiv & \lambda l.\, \mathbf{if}\, l = l_j \;\mathbf{then}\; x \;\mathbf{else}\; r \cdot l \end{array}$$

The record type is translated to a *dependent* function type. We use the notation $x \colon A \rightarrow B_x$ to specify the functions that have type $B[a/x]$ on any argument $a \in A$.

$$\{l_i \colon T_i{}^{i \in \{1 \ldots n\}}\} \quad \equiv \quad l \colon \mathbf{Atom} \rightarrow \mathbf{case}\, l \;\mathbf{of}\; l_1 \colon T_1 | \cdots | l_n \colon T_n | \_\colon \mathbf{Top}$$

2.2 Dependent records

When we develop the interpretation for formal objects, we will also need *dependent* records to describe objects with more complete specifications. For example, a complete specification of the point object includes a specification of the relation between *getX* and *setX*, as follows:

$$\begin{array}{ll} \mathit{Point}' \equiv \exists \mathit{Rep} \colon \mathbf{Type}.\{ & \mathit{state} \colon \mathit{Rep}; \\ & \mathit{methods} \colon \{ \; \mathit{getX} \colon \mathit{Rep} \rightarrow \mathbb{Z}; \\ & \qquad \mathit{setX} \colon \mathit{Rep} \rightarrow \mathbb{Z} \rightarrow \mathit{Rep}; \\ & \qquad \mathit{spec} \colon \forall r \colon \mathit{Rep}, i \colon \mathbb{Z}. \mathit{getX}(\mathit{setX}(r)(i)) = (\mathit{getX}(r)) + i\}\} \end{array}$$

In order to encode dependent records, we need a function type where dependencies are allowed to range over several values of the function. Given such a function type, the type for the methods of *Point′* would be encoded as follows:

$$\begin{array}{ll} \mathit{methods} \colon (l \colon \mathbf{Atom} \rightarrow & \mathbf{case}\, l \;\mathbf{of} \\ & \mathit{getX} \colon \mathit{Rep} \rightarrow \mathbb{Z} \\ & \mathit{setX} \colon \mathit{Rep} \rightarrow \mathbb{Z} \rightarrow \mathit{Rep} \\ & \mathit{spec} \colon \forall r \colon \mathit{Rep}, i \colon \mathbb{Z}.\; (\mathit{methods}(\mathit{getX}))((\mathit{methods}(\mathit{setX}))(r)(i)) \\ & \qquad\qquad\qquad = ((\mathit{methods}(\mathit{getX}))(r)) + i \\ & \_\colon \mathbf{Top} \end{array}$$

We develop this dependent function type (called the *very dependent function type*) in the following section. Afterwards, we develop the interpretation of objects in the type theory, covering the issues of subtyping and method subsumption. In Section 5, we use the property of propositions–as–types to reflect the account of objects back into the type theory to provide a foundation for formal abstract data types and objects.

3 Very-dependent function types

In order to represent dependent records types as dependent function types, we need to allow the dependencies to refer to *values* of the functions. For instance, in the record type

$$\{T\colon \mathbf{Type}; getX\colon T \to \mathbb{Z}; setX\colon T \to \mathbb{Z} \to T\},$$

the types for *getX* and *setX* refer to the value for the field T. We can describe this type as a function type by allowing a representative of the function to be used to describe its range. We use the following notation,

$$\{f \mid x\colon A \to B_{f,x}\},$$

to specify the functions f with domain type A, and range $B_{f,a}$ on any particular argument $a \in A$. For instance, the type

$$\{f \mid i\colon \mathbb{N} \to \texttt{if}\, i > 0 \,\texttt{then}\, \{f(i-1)+1\ldots\} \,\texttt{else}\, \mathbb{N}\}$$

denotes the monotonically increasing functions in $\mathbb{N} \to \mathbb{N}$. Of course, some of these types may not be sensible, for example $\{f \mid i\colon \mathbb{N} \to (f(i) \to f(i))\}$. The difficulty with this example is the use of a cycle to describe the value of the function, i.e. the type for $f(i)$ depends on the value $f(i)$ itself. To avoid these constructions, we require that the type specification for the range respect some well-founded order on the domain. This restriction enables a well-founded semantic description of the type, which in turn preserves the predicative construction of the type theory. However, at this time it is unclear whether a predicative semantic account can be given that preserves the intuitive meaning of the very-dependent function type and also allows the occurrence of cycles in the type description.

The semantic account we provide for the very-dependent function types is based on the predicative PER semantics of Allen [2]. In Allen's framework, the semantics for a term T is described by defining when T is a type and specifying what its equivalence relation on members is. Since the type theory is specified as a fixed-point, typehood judgments are only allowed to occur positively in the semantic definition. The complete semantics for the very-dependent functions $\{f \mid x\colon A \to B_{f,x}\}$ are as follows:

Semantics 1 *The term $\{f \mid x\colon A \to B\}$ is a type with equality φ, if and only if:*

1. *A is a type with equality α,*
2. *A is well-founded with respect to some partial order $<$,*
3. *for any a such that $\alpha(a,a)$, there are relations γ_a and $\beta_{g,a}$ satisfying the following conditions:*
   - *(a) $\{f \mid x\colon \{a'\colon A \mid a' < a\} \to B\}$ is a type with equality γ_a,*
   - *(b) $B[g,a/f,x]$ is a type with equality $\beta_{g,a}$ for any g where $\gamma_a(g,g)$,*
4. *for terms f and f', $\varphi(f,f')$ if, and only if, for all a' satisfying $\alpha(a',a')$, $\gamma_a(f,f')$ and $\beta_{f,a'}(f(a'),f'(a'))$.*

$$\dfrac{\begin{array}{ll} 1. & A \in \mathbb{U}_i \\ 2. & < \;\in A \rightarrow A \rightarrow \mathbb{U}_i \\ 3. & \mathit{WellFounded}_i\,(A; <) \\ 4. & H, y\colon A, g\colon \{f \mid x\colon \{z\colon A \mid z < y\} \rightarrow B\} \vdash B[g, y/f, x] \in \mathbb{U}_i \end{array}}{H \vdash \{f \mid x\colon A \rightarrow B\} \in \mathbb{U}_i} \text{ Typing (some } <)$$

$$\dfrac{\begin{array}{ll} 1. & H \vdash \{f \mid x\colon A \rightarrow B\} \in \mathbb{U}_i \\ 2. & H, b\colon A \vdash b_1[b/x_1] = b_2[b/x_2] \in B[\lambda x_1.b, b/f, x] \end{array}}{H \vdash \lambda x_1.b_1 = \lambda x_2.b_2 \in \{f \mid x\colon A \rightarrow B\}} \in$$

$$\dfrac{\begin{array}{ll} 1. & f_1 \in \{g \mid x\colon C \rightarrow D\} \quad \text{(some } C,\ D) \\ 2. & f_2 \in \{g \mid x\colon E \rightarrow F\} \quad \text{(some } E,\ F) \\ 3. & H, y\colon A \vdash f_1(y) = f_2(y) \in B[f_1/g] \end{array}}{H \vdash f_1 = f_2 \in \{g \mid X\colon A \rightarrow B\}} \in \text{(extensional)}$$

$$\dfrac{\begin{array}{ll} 1. & H, f\colon \{g \mid x\colon A \rightarrow B\}, J \vdash a \in A \\ 2. & H, f\colon \{g \mid x\colon A \rightarrow B\}, J, y\colon B[f, a/g, x], y = f(a) \in B[f, a/g, x] \vdash T \quad \text{ext } t \end{array}}{H, f\colon \{g \mid x\colon A \rightarrow B\}, J \vdash T \quad \text{ext } t[f(a)/y]} \text{ Elim}$$

$$\dfrac{\begin{array}{ll} 1. & A \in \mathbb{U}_i \\ 2. & <\in A \rightarrow A \rightarrow \mathbb{U}_i \\ 3. & \mathit{WellFounded}_i\,(A; <) \\ 4. & H, y\colon A, g\colon \{f \mid x\colon \{z\colon A \mid z < y\} \rightarrow B\} \vdash B[g, y/g, x] \quad \text{ext } b \end{array}}{H \vdash \{f \mid x\colon A \rightarrow B\} \quad \text{ext } \lambda y.Y(\lambda g.b)} \text{ Form (some } <)$$

$$\dfrac{H \vdash a_1 = a_2 \in A \quad H \vdash f_1 = f_2 \in \{f \mid x\colon A \rightarrow B\}}{H \vdash f_1(a_1) = f_2(a_2) \in B[f_1, a_1/f, x]} \text{ Apply} =$$

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

This semantics is well-defined because of the well-foundedness restriction on the domain type A. The membership predicates γ and β can be constructed by induction. In the base case, the type $\{f \mid x\colon \{a''\colon A \mid a'' < a'\} \rightarrow B\}$ reduces to $\{f \mid x\colon \mathbf{Void} \rightarrow B\}$ which is a trivial type for any term B. Similarly, the semantics at the base case require that $B[g, a']$ be a type for the function g with domain **Void**, which means that B cannot evaluate g on any argument. The definitions for γ and β are constructed mutually recursively for the entire domain A.

The rules extracted from the semantics are shown in Table 1. The most interesting rules are for type membership, extensional equality, and for type formation. Membership requires a witness for the well-founded order $<$ for the domain A, but the use of the $<$ relation is local to the membership proof, not the type. It is possible to use several different well-orders to generate distinct proofs of membership.

Two very-dependent functions are extensionally equal if they compute the same results when given the same arguments (the requirements $f_1 \in \{f \mid x\colon C \rightarrow D\}$ and $f_2 \in \{f \mid x\colon E \rightarrow F\}$ mean that f_1 and f_2 must be functions). The extensional behavior is the crucial step in deriving the standard subtyping behavior of the function type, where

$$\{f \mid x{:}\, A' \to B'\} \subseteq \{f \mid x{:}\, A \to B\}$$

if

$$A' \supseteq A \quad \text{and} \quad \forall a{:}\, A, g{:}\, \{f \mid x{:}\, A|_a^< \to B\}.B'_{g,a} \subseteq B_{g,a},$$

where we use the notation $A|_a^<$ to specify the elements of A that are smaller than a according to the relation $<$.

Finally, note that the formation rule contains a fixed point in the extracted proof. This proof extract is known to normalize because of the well-founded order on the domain.

3.1 Dependent records

We can encode dependent records as very-dependent function types with the following translation. Let R be the record type $R = \{l_i{:}\, M_i \ ^{i\in\{1\ldots n\}}\}$, where M_i may contain occurrences of the labels $l_1, \ldots, l_{i-1}$. The encoding of R is as follows:

$$R = \{f \mid l{:}\, \textbf{Atom} \to \textbf{case}\, l\, \textbf{of} \cdots |l_i{:}\, M_i[f(l_j)/l_j \ ^{i\in\{1\ldots i-1\}}]\}$$

The following table summarizes the encoding of records and their types.

For the record $r = \{l_i = m_i \ ^{i\in\{1\ldots n\}}\}$		
r	$=$	$\lambda l.\, \textbf{case}\, l\, \textbf{of} \cdots |l_i{:}\, m_i| \cdots |\_{:}\, \textbf{Top}$
$r \cdot l_i$	$=$	$r(l_i)$
$r \cdot l_i := m$	$=$	$\lambda l.\, \textbf{if}\, l = l_i\, \textbf{then}\, m\, \textbf{else}\, r \cdot l$
$\{l_i{:}\, M_i \ ^{i\in\{1\ldots n\}}\}$	$=$	$\{f \mid l{:}\, \textbf{Atom} \to \textbf{case} \cdots |l_i{:}\, M_i[f(l_j)/l_j \ ^{i\in\{1\ldots i-1\}}]\}| \cdots |\_{:}\, \textbf{Top}$

The rules for this record calculus can be derived from the rules of the very-dependent type and from the rules of the type theory. These record rules are show in Table 2. These rules have the standard feel of a record calculus, but the rule for field override deserves special attention. When a field is overridden, it may change the typing for *other* fields. In general, it may not be possible to give a typing for a single override. For example, consider the record describing points on the unit circle:

$$\{x{:}\, \mathbb{R}; y{:}\, \mathbb{R}; \_{:}\, x^2 + y^2 = 1\}$$

An update for a record of this type will most likely have to update the values of x and y simultaneously because an intermediate form would not have the desired type. The rule we give for update, check that the remaining fields are all well-typed. A more general rule for update could be derived for multiple simultaneous updates if desired.

4 Object Interpretation

The record calculus presented in the previous section provides the basis for an existential interpretation of objects. Following the account of Pierce and Turner [23], we require that objects have method descriptions parameterized by the type of the internal state representation. For instance, the method description for the *Point* object is:

$$PointM \equiv \lambda Rep.\{getX{:}\, Rep \to \mathbb{Z}; setX{:}\, Rep \to \mathbb{Z} \to Rep\}$$

$$
\begin{array}{|l|}
\hline
\begin{array}{lrcl}
\text{Let} & r & \equiv & \{l_i = m_i \ ^{i\in\{1\ldots n\}}\} \\
& R & \equiv & \{l_i\colon M_i \ ^{i\in\{1\ldots n\}}\}
\end{array} \\[2ex]
\dfrac{\Gamma, l_1\colon M_1, \ldots, l_{i-1}\colon M_{i-1} \vdash M_i \in \mathbf{Type} \quad (\text{for } i \in \{1\ldots n\})}{\Gamma \vdash R \in \mathbf{Type}} \ \text{typing} \\[2ex]
\dfrac{\Gamma \vdash r.l_i \in M_i[r \cdot l_j / l_j \ ^{j\in\{1\ldots i-1\}}] \quad (\text{for } i \in \{1\ldots n\})}{\Gamma \vdash r \in R} \ \text{mem} \\[2ex]
\dfrac{\Gamma \vdash r \in R}{\Gamma \vdash r \cdot l_i \in M_i[r \cdot l_j / l_j \ ^{j\in\{1\ldots i-1\}}]} \ \text{sel} \\[2ex]
\dfrac{\begin{array}{ll}
1. & r \cdot l_j \in M_i[r \cdot l_k / l_k \ ^{k\in\{1\ldots j-1\}}] \quad (\text{for } j \in \{1\ldots i-1\}) \\
2. & m \in M_i[r \cdot l_j / l_j \ ^{j\in\{1\ldots i-1\}}] \\
3. & r \cdot l_j \in M_i[r \cdot l_k / l_k \ ^{k\in\{1\ldots j-1\}-\{i\}, m/l_i}] \quad (\text{for } j \in \{i+1\ldots n\})
\end{array}}{\Gamma \vdash r \cdot l_i := m \in R} \ \text{over} \\
\hline
\end{array}
$$

Table 2: Rules for dependent records

This description is parameterized over possible state representations *Rep*. In general, method descriptions will belong to $\mathbf{Type} \to \mathbf{Type}$.

A generic *Object* constructor can be defined to produce the type for an object from its method description as follows:

$$
\mathit{Object} \equiv \lambda M.\exists \mathit{Rep}\colon \mathbf{Type}.\{\mathit{state}\colon \mathit{Rep}; \mathit{methods}\colon M(\mathit{Rep})\}
$$

For instance, the *Point* type is constructed by applying the *Object* constructor to the method description *PointM*:

$$
\begin{array}{rcl}
\mathit{Point} & \equiv & \mathit{Object}(\mathit{PointM}) \\
& = & \exists \mathit{Rep}\colon \mathbf{Type}.\{\ \mathit{state}\colon \mathit{Rep}; \\
& & \qquad \mathit{methods}\colon \{\ \mathit{getX}\colon \mathit{Rep} \to \mathbb{Z}; \\
& & \qquad\qquad \mathit{setX}\colon \mathit{Rep} \to \mathbb{Z} \to \mathit{Rep}\}\}
\end{array}
$$

New *Point* objects are created by recording the state and the methods. For instance, a point at the location 0 is described by the following object:

$$
\begin{array}{rcl}
p_0 & \equiv & \texttt{pack}\{\ \mathit{state} = 0; \\
& & \qquad \mathit{methods}\colon \{\ \mathit{getX} = \lambda x.x; \\
& & \qquad\qquad \mathit{setX} = \lambda x, i.x + i\}\}
\end{array}
$$

where $p_0 \in \mathit{Point}$.

This specification differs from the standard account because the point object is untyped. The `pack` operation does not actually save the type being injected into the weak sum. However, in order to prove that the point p_0 is a *Point*, it is necessary to provide a witness for the existential at proof time.

The final step in the construction is to specify wrappers for the methods to encode the self-applicative behavior of objects. These wrappers serve as the standard external interface to the object, and are constructed by extracting the state and methods from the argument object, applying the method to the state, the repacking the object. For example, the *setX* method is invoked as follows:

$$Point`setX = \lambda p, i.\, \mathtt{open}\, p\ \mathtt{as}\ r\ \mathtt{in}\ \mathtt{pack}(r \cdot state := (r \cdot methods \cdot setX)\ (r \cdot state)\ i)$$

where $Point`setX \in Point \rightarrow \mathbb{Z} \rightarrow Point$.

4.1 Subtyping

The typing $Point \rightarrow \mathbb{Z} \rightarrow Point$ for the $Point`setX$ wrapper is fairly restrictive. For example, consider the type of colored points, which have a color in addition to a position. The method description for colored points is as follows:

$$\begin{aligned} CPointM = \lambda Rep.\{\ & getX\colon Rep \rightarrow \mathbb{Z};\\ & setX\colon Rep \rightarrow \mathbb{Z} \rightarrow Rep;\\ & getC\colon Rep \rightarrow Color;\\ & setC\colon Rep \rightarrow Color \rightarrow Rep\} \end{aligned}$$

Since the record type for colored point is an extension of the record type for *Point*, the $Point`setX$ method can be applied to a *CPoint*. However, it "forgets" the extra fields in the colored point and returns a *Point*, which is most likely an undesired result.

The problem is not in the implementation of $Point`setX$, it is in its typing judgment. Since the $Point`setX$ wrapper refers only to those parts of the object that are present in point (and since it doesn't modify other fields), it is actually polymorphic over all subobjects that allow the *state* to be updated.

We introduce the relation $\preceq$ to describe the desired class of subobjects. We intend that $A \preceq B$ if A has all the methods of B, and possibly more. We can describe this behavior by qualifying the method descriptions:

$$\begin{aligned} A \preceq B \quad \text{iff} \quad & \exists AM, BM\colon \mathbf{Type} \rightarrow \mathbf{Type}.\\ & \quad A = Object(AM) \wedge B = Object(BM) \wedge \forall T\colon \mathbf{Type}.AM\ T \subseteq BM\ T\\ & \quad \wedge \forall b\colon B.\exists a\colon A.a = b \in B \end{aligned}$$

The final part of this description states that each element in the the supertype has a corresponding representation in the subtype. This requirement is a stronger version of the "positive subtyping" of Hofmann and Pierce [15]. In their formulation, the rules for subtyping are constrained so that *all* subtypes must obey this "update" restriction. In our case, subtyping retains its original mathematical meaning, but if $A \preceq B$, then $A \subseteq B$. Our version is stronger because we also require that the representation types of subobjects be subtypes. A more complete treatment of objects would require a more extensional definition of equality over the *behavior* of objects, excluding their states and representation types.

A similar subtyping restriction is given by Bruce [8], who addresses this problem in PolyTOIL by defining "matching," where an object "matches" another if any method that can be invoked on the first object can also be invoked on the second.

Given this subtyping judgment, the subtyping criterion for $Point`setX$ can be made more explicit:

$$\forall A \preceq Point.Point`setX \in A \rightarrow \mathbb{Z} \rightarrow A$$

or, equivalently,

$$Point`setX \in \bigcap A \preceq Point.(A \to \mathbb{Z} \to A)$$

Note that the $Point`setX$ wrapper is not polymorphic over *all* subtypes of $Point$ — only those that are objects with more methods. No criterion is specified for the representation type.

4.2 Generalized system

The tools used to construct this object system can be generalized somewhat. Consider a general method description:

$$M = \lambda Rep.\{l_i = M_i \;^{i \in \{1 \ldots n\}}\}$$

Object types are constructed by applying the *Object* constructor to the method description.

$$Object \equiv \lambda M. \exists Rep\colon \mathbf{Type}.\{state\colon Rep; methods\colon M(Rep)\}$$

Objects are constructed by specifying the representation type Rep, as well as specifying methods $m_1, \ldots, m_n$ that inhabit the types $M_1, \ldots, M_n$. We package this construction using the following definition:

$$object(s, l_1 = m_1, \ldots, l_n = m_n) \equiv \{state = s, methods = \{l_i = m_i \;^{i \in \{1 \ldots n\}}\}\}$$

For method wrappers, we distinguish between methods that return a modified object, and those that don't. Consider a method m_i of the former type, of arity j. Let $\overline{a}$ be a list of length j of fresh variables. Then the wrapper for method m_i is defined as follows:

$$o.l_i \equiv \texttt{open}\, o \;\texttt{as}\; r \;\texttt{in}\; \lambda\overline{a}.\,\texttt{pack}(r \cdot state := (r \cdot methods \cdot l_i)\ (r \cdot state)\ \overline{a})$$

For methods that do not return a modified object, the wrapper has a simpler implementation. Let m_j be a method that does not return a modified object. Then

$$o.l_j \equiv \texttt{open}\, o \;\texttt{as}\; r \;\texttt{in}\; (r \cdot methods \cdot l_i)\ (r \cdot state)$$

Given these definitions, the rules for the object calculus are listed in Table 3. There is no rule for method override because all updates to the object occur through method selection. The most interesting rules are for method selection. We provide rules two cases: the first is for the case where the Rep type occurs only as in the first argument position of M_i, i.e. $M_i = Rep \to M_i'$ and Rep does not occur in M_i'. In this case, the type M_i' completely describes the type of method. In the second case, the method is allowed to return an updated object, but Rep is not allowed to occur negatively in M_i'. In this case, since the wrapper repacks the object after calling the method, the final result has the type O.

The case we do not consider for method selection is the case where the representation type occurs negatively in the method type (except as the type fo the first argument). This rules out the handling of binary methods on the representation type. Such binary methods are problematic in any case, because the representation type is specific to each object. Since a method in one object doesn't have any information about the representation type for any other object, it is not possible to write binary methods of this form. A more complete discussion of binary methods is given in Bruce et.al. [6].

Proper handling of binary methods requires the use of an explicit representation type, as we describe in the following section for a description of monoids. Exposing the representation type results in a dependent object, which once again requires the use of an encoding using very-dependent types.

$$
\begin{array}{llll}
\text{Let} & M & \equiv & \lambda Rep.\{l_i = M_i \ ^{i\in\{1\ldots n\}} \quad (\text{where } M_i = Rep \to M_i') \\
& O & \equiv & Object(M) \\
& o & \equiv & object(s, l_1 = m_1, \ldots, l_n = m_n)\}
\end{array}
$$

$$
\frac{\Gamma, Rep\colon \mathbf{Type}, l_1\colon M_1, \ldots, l_{i-1}\colon M_{i-1} \vdash M_i \in \mathbf{Type} \quad (\text{for } i \in \{1\ldots n\})}{\Gamma \vdash O \in \mathbf{Type}}\ \text{typing}
$$

$$
\frac{\Gamma \vdash m_i \in M_i[m_j/l_j \ ^{j\in\{1\ldots i-1\}}, s/Rep] \quad (\text{for } i \in \{1\ldots n\})}{\Gamma \vdash o \in O}\ \text{mem}
$$

$$
\frac{\Gamma \vdash o \in O \quad (O \text{ does not occur in } M_i')}{\Gamma \vdash o \cdot l_i \in M_i[m_j/l_j \ ^{j\in\{1\ldots i-1\}}]}\ \text{sel}_1
$$

$$
\frac{\Gamma \vdash o \in O \quad (O \text{ occurs positively in } M_i')}{\Gamma \vdash o \cdot l_i \in M_i[m_j/l_j \ ^{j\in\{1\ldots i-1\}}, O/Rep]}\ \text{sel}_2
$$

Table 3: Rules for object calculus

5 Theories as Objects

The encoding of objects as dependent records in the type theory provides a great deal of expressive power, which can be put to use in the type theory to provide tools for formal modularity with inheritance.

Since the method values can also be used in method types the propositions–as–types correspondence allows us to express precise constructions. For example, the following example describes points on the unit circle, with a rotation function.

$$
\begin{array}{ll}
CirclePointM \equiv \{ & getX\colon Rep \to \mathbb{R}; \\
& getY\colon Rep \to \mathbb{R}; \\
& spec\colon r\colon Rep \to (getX\ r)^2 + (getY\ r)^2 = 1; \\
& rot\colon Rep \to \mathbb{R} \to Rep\}
\end{array}
$$

These same techniques can be put to use in the domain of abstract algebra to provide type-theoretic support for relations between algebraic objects. For instance, a monoid can be described with the following signature:

$$
\begin{array}{ll}
MonoidM \equiv \{ & car\colon \mathbf{Type}; \\
& \oplus\colon car \to car \to car; \\
& \approx\colon car \to car \to \mathbf{Prop}; \\
& e\colon car; \\
& assoc\colon \forall x, y, z\colon car.(x \oplus y) \oplus z \approx x \oplus (y \oplus z); \\
& left\_unit\colon \forall x\colon car.x \oplus e \approx x; \\
& right\_unit\colon \forall x\colon car.e \oplus x \approx x\}
\end{array}
$$

Note that this definition of monoids uses a *dependent* object type (in fact, there are multiple dependencies for many of the methods). The *car* is the carrier for elements in the Monoid, and the function $\oplus$ is a binary method on elements of in the carrier. Given

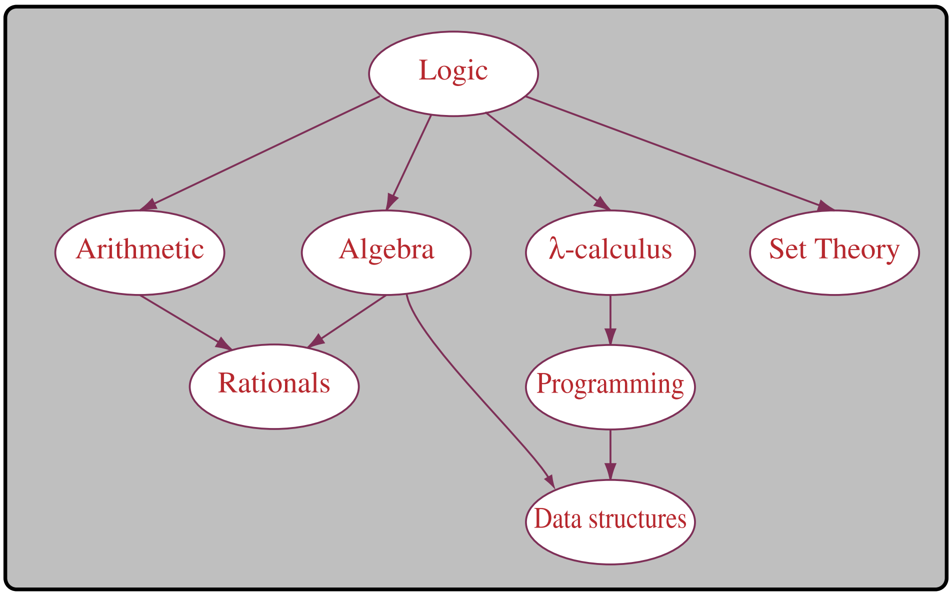

Figure 1: Possible Theory Organization

this definition, it is possible to derive many properties of general Monoids as theorems in the type theory. Theorems about specific monoids can be derived for elements of the type *Monoid*.

A Group is a simple extension of a Monoid, with an additional requirement that each element of the carrier have an inverse.

$$
\begin{aligned}
GroupM \equiv \{\ & car\colon \mathbf{Type};\\
& \oplus\colon car \to car \to car;\\
& \approx\colon car \to car \to \mathbf{Prop};\\
& e\colon car;\\
& assoc\colon \forall x, y, z\colon car.(x \oplus y) \oplus z \approx x \oplus (y \oplus z);\\
& left\_unit\colon \forall x\colon car.x \oplus e \approx x;\\
& right\_unit\colon \forall x\colon car.e \oplus x \approx x;\\
& inverse\colon \forall x\colon car.\exists y\colon car.x \oplus y \approx e\}
\end{aligned}
$$

Since the object type *Group* is a subtype of *Monoid*, all the theorems for Monoids can be inherited by the group, in the same manner that methods are subsumed by subobjects.

The propositions–as–types correspondence reveals an unexpected benefit. The type *Monoid* is also a proposition, the proof of which is a specific monoid object. The standard proof techniques can be used to produce objects within an object type. Although this is not so useful in most programming domains (where, for instance, the proofs of *Point* and *ColorPoint* are numerous and trivial) perhaps the biggest payoff is in the framework for type theories themselves. Just as the axioms for a monoids can be expressed within the method description for monoids, the rules for a type theory can be expressed within the module defining the type theory. A possible organization of modules in a formal system is shown in Figure 1.

Currently we are using these techniques to build a new version of the Nuprl system, with support for object-oriented construction of formal modules ("theories"). Theories are stated as object signatures, and proofs of theories are objects that inhabit the theory type.

The formalization of the type theory is itself a collection of object types, which allows the treatment of the type theory itself as a formal object.

6 Conclusion

We have presented a formal interpretation of objects in type theory, providing a a solid mathematical interpretation of object-oriented programming, as well as providing a new framework for formal object-oriented modules. The very-dependent type serves as the link between these two worlds. Dependent formal objects types are represented as very-dependent function types, expanding the expressivity of the formalization of objects, and allowing the use of objects to describe the type theory itself. The proposition-as-types correspondence provides a meaning for modules in the type theory as "theories" or "sub-domains." Dependent object types also allow the natural construction of binary methods that use explicit representation types.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, Forthcoming.

[2] Stuart F. Allen. *The Semantics of Type Theoretic Languages.* PhD thesis, Cornell University, August 1986.

[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] David Aspinall and Adriana Compagnoni. Subtyping dependent types. Unpublished at this time (available at `http://www.dcs.ed.ac.uk/home/da/psub.ps.gz`).

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

[6] Kim Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1996. To appear. Also available electronically and as a technical report from ENS, DEC SRC, Williams, and Iowa State.

[7] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing, and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[8] Kim B. Bruce and Leaf Peterson. Subtyping is not a good "Match" for object-oriented languages. Technical report, Williams College, 1996. (extended abstract).

[9] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type Theory and Programming. *EATCS*, February 1994. bulletin no 52.

[10] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. `http://www.csl.sri.com/sri-csl-fm.html`.

[11] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The coq proof assistant user's guide. Technical Report Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[12] R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System.* Prentice–Hall, 1986.

[13] 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.

[14] Jason J. Hickey. A semantics of objects in type theory. Online formalization available through Cornell CS.

[15] Martin Hofmann and Benjamin Pierce. Positive subtyping. In *Proceedings of Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 186–197. ACM, January 1995. Full version available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994. To appear in Information and Computation, 1996.

[16] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In *AMAST '96*, 1996.

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

[18] Zhaohui Luo and Randy Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-202, University of Edinburgh, 1992.

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

[20] Per Martin-Löf. *Intutionistic Type Theory.* Bibliopolis, Napoli, 1984.

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

[22] 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.

[23] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[24] Bernhard Reus and Thomas Streicher. Verifying Properties of Modules Construction in Type Theroy. In *18th International Symposium on Mathematical Foundations of Computer Science*, pages 660–670. Springer, Aug/Sept 1993.

[25] Adrian Rezus. Semantics of constructive type theory. Technical Report 70, Nijmegen University, The Netherlands, September 1985.

[26] Donald T. Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

A Type Theory Fragment

We only use a small fragment of the type theory in this account of objects, but this fragment is quite powerful. A significant fraction of the type theory can be reduced to the very-dependent function type.

The programs are untyped, and include the terms of the untyped lambda calculus, numbers, and strings.

$$
\begin{array}{rcl}
\textbf{Terms} & ::= & x, f(a), \lambda x.b, \\
& & \{\ldots, -2, -1, 0, 1, 2, \ldots\}, \\
& & \text{“}a\text{”}, \text{“}b\text{”}, \ldots, \\
& & \textbf{if}\ a = b\ \textbf{then}\ t_1\ \textbf{else}\ t_2, \\
& & \textbf{case}\ a\ \textbf{of}\ l_1 \colon t_1 | \cdots | l_n \colon t_n, \\
& & \mathit{Types}
\end{array}
$$

The **if** term reduces to t_1 if a and b are equal integers or strings, or t_2 if they are not equal. The **case** construct is defined in terms of **if**.

Types are also terms, and include the the following:

$$
\begin{array}{rcl}
\textbf{Types} & ::= & \textbf{Void}, \mathbb{Z}, \textbf{Atom}, \mathbb{U}_k, a = b \in T, \\
& & A \subseteq B, \{x \colon A \mid B_x\}, \\
& & x \colon A \to B_x, \bigcap x \colon A.B_x, \{f \mid x \colon A \to B_{f,x}\}, \textbf{Top}
\end{array}
$$

The Nuprl type theory provides a consistent correspondence between propositions and types. Every type is a proposition, and the proposition is true if the type is inhabited. In that case, the inhabitant is the *proof* of the proposition.

The type **Void** is empty, and the types $\mathbb{Z}$ and **Atom** contains the numbers and strings, respectively. The equality type $a = b \in T$ is inhabited (true) if a and b are terms in the type T, and $a = b$ in the equality for T. The subtype-type $A \subseteq B$ is inhabited if A and B are types, and for any two elements $x, y \in A$, if $x = y \in A$ then $x = y \in B$. The “set” type $\{a \colon A \mid B_a\}$ is inhabited by elements $a \in A$ where B_a is true. Note that for a type A, there is a trivial proof of $\{x \colon A \mid B_x\} \subseteq A$ for any type B_x.

Types are organized into type *universes* $\mathbb{U}_k$ for $k \in \{1, 2, \ldots\}$. For all the types we consider here, a type is an element of $\mathbb{U}_k$ if each of its type subformulas are. In addition, $\mathbb{U}_k \in \mathbb{U}_{k+1}$. Theorems are usually stated relative to an arbitrary type universe, say $\mathbb{U}_k$, where k is free, and is called a level-variable. Free occurrences of level-variables in a theorem are all universally quantified.

The remaining types are all essentially dependent function types. The dependent function type $x \colon A \to B_x$, also written $\Pi x \colon A.B_x$, contains the functions f where $f(a) \in B[a/x]$ for any element $a \in A$. The intersection type $\bigcap x \colon A.B_x$ contains the elements b where $b \in B_a$ for every $a \in A$. In some sense this type is inhabited by *constant* functions in $x \colon A \to B_x$. The type **Top** is defined as the intersection $\bigcap x \colon \textbf{Void}.\textbf{Void}$, which degenerates to the type of all terms. The very-dependent function type $\{f \mid x \colon A \to B_{f,x}\}$ contains the functions g where $g(a) \in B[g, a/f, x]$ for any $a \in A$. In this case, the type A must be well-founded, and the type $B[g, a/f, x]$ can only apply g to values $a' \in A$ where $a' < a$.

The weak existential type is encoded as the type of its projection function (`open`). This has the encoding:

$$\exists a\colon A.B_a \equiv \bigcap_{T:\mathbf{Type}} \left(\bigcap_{a:A} (B_a \to T) \right) \to T$$

The intuition behind this construction is that the projection function takes as its body a function that is polymorphic over all types B_a for any representation type $a \in A$. Given a polymorphic body that returns a value of type T, and a packed object of type B_a for some $a \in A$, the `open` function returns a value of type T.

The encoding of `pack` and `unpack` are as follows:

$$\begin{array}{rcl} \texttt{pack}(x) & \equiv & \lambda f.f(x) \\ \texttt{open}\, o\, \texttt{as}\ r\ \texttt{in}\ b_r & \equiv & o\ (\lambda r.b_r) \end{array}$$

where $\texttt{pack}(x) \in \exists a\colon A.B_a$ if there is an $a \in A$ such that $x \in B_a$, and $\texttt{open}\, o\, \texttt{as}\ r\ \texttt{in}\ b_r \in T$ if $o \in \exists a\colon A.B_a$ and $\lambda y.b_y \in \bigcap a\colon A.(B_a \to T)$.