A Predicative Type-Theoretic Interpretation of Objects

Jason J. Hickey
Department of Computer Science
Cornell University

Abstract

Predicative type theories are powerful tools for giving foundational interpretations of programming languages. Due to their explicit inductive construction, predicative type theories have multiple mathematical models that provide precise definitions of programming language features. However, not all features have predicative interpretations, and current interpretations of objects rely on impredicative type theories, such as Girard's System F, because of the difficulty in specifying a type for objects in the presence of self-application.

In this paper we show that objects have a predicative interpretation. We show that predicativity is associated with method monotonicity, and that binary methods prevent the inductive type construction. Our interpretation differs from impredicative accounts by replacing the use of recursive types for objects with conditions for method polymorphism over the *self* type. We further give a propositional meaning to objects in the type theory, providing a calculus for formal verification. Our interpretation has been verified in the Nuprl predicative type theory.

1 Introduction

Object-oriented languages provide mechanisms for program modularity and reuse that simplify the task of programming enough that object-oriented programming is now common practice. Unfortunately, the theory of objects has lagged behind practice due, in part, to the difficulty of assigning a type to self-application in the presence of method override. Self application occurs during method selection when the entire object is passed to the method as an implicit "self" argument.

Recent type-theoretic accounts of objects [1, 21, 5, 10] have been encoded in impredicative type theory (mainly variants of Girard's System F [9]), which allow self-application through an impredicative interpretation of the self parameter. In this paper, we show that the construction of objects has an interpretation in *predicative* type theory.

This is a stronger result because predicative theories make fewer assumptions and have more models. In particular, the Nuprl type theory that we use in this paper has multiple mathematical models, notably set theoretic [17], PER models [3, 20], denotational models [23], and others. In addition, our interpretation is extendable to foundational formal objects that encode *proofs* [15, 11], a critical step toward our goal of providing a modular and object-oriented logical framework [12].

The object interpretation is more delicate but we get more information. For example, predicative theories have a strictly inductive construction [19, 4], so the account of objects is inductive. An interesting feature is that impredicative type recursion is replaced by method polymorphism. The explicit inductive construction is also valid in impredicative type theory.

Our interpretation has the following benefits.

- The predicative account provides a simple, precise, mathematical account of objects.
- Inheritance is rooted in method polymorphism; recursive types are not necessary.
- We rely on the formal foundations of the type theory to verify soundness of the interpretation. A formal proof of soundness is available online.

In our interpretation, object types have propositional meaning. We further extend the calculus with dependent method types to provide a basis for formal program analysis. We develop our account on the object calculus of Abadi et al. [1, 2], which gives a precise characterization of self application and method override. We show that the type system can be developed without the use of recursive types or weak sums, and without sacrificing subtyping properties. Method type dependencies are expressed using dependent records based on the very-dependent function type [11], and method polymorphism over subtypes is expressed using intersection types.

2 Review

The formalism we use for objects in this paper is a functional object calculus similar to the calculus of Abadi, Cardelli, and Viswanathan [2], in which the only primitives are objects, method invocation, and method update. While these object calculi are simple and precise, they allow representations of class-based and object-based concepts, as shown by Abadi and Cardelli in the "Theory of Objects" [1]. One of the significant features of these calculi is that they are all based on

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, from DARPA through grant 93-11-271, and from AASERT through grant N00014-95-1-0985.

Values	v	$::=$	$[l_i = \lambda x_i.t_i]^{\ i\in\{1\ldots n\}}$
Terms	t	$::=$	$v \mid t.l \mid t.l \Leftarrow \lambda x.t$
Types	T	$::=$	$Obj(X.[l_i{:}\, T_i[X]^{i\in\{1\ldots n\}}])$

Figure 1: Syntax of the object calculus

self-application: methods receive the object as an argument when they are invoked.

Since we are giving a semantic characterization of programs, we use an untyped operational calculus that allows programs to be interpreted according to their context. The syntax of the source language is given in Figure 1. In this language, objects $o = [l_i = \lambda x_i.t_i]^{\ i\in\{1\ldots n\}}$ are finite collections of *methods* labelled with names $l_1, l_2, \ldots, l_n$ drawn from an infinite collection of names. The method bodies $\lambda x_i.t_i$ are functions, where the variable x_i is a binding occurrence representing the "self" parameter for the method named l_i (Abada et. al. use the notation $\varsigma x_i.t_i$ to emphasize the difference between method bodies and standard lambda abstraction).

There are two operations on objects. *Method selection*, written $o.l_j$, substitutes the object o for x_j in body t_j. *Method override*, written $o.l_j \Leftarrow \lambda x.t$, creates a new object from o, replacing the method $\lambda x_j.t_j$ with the new method $\lambda x.t$. This operational semantics is summarized in the following table:

For an object $o = [l_i = \lambda x_i.t_i]^{\ i\in\{1\ldots n\}}$:

$$\begin{array}{rcl} o.l_j & \rightarrow & t_j[o/x_j] \\ o.l_j \Leftarrow \lambda x.t & \rightarrow & [l_j = \lambda x.t, l_i = \lambda x_i.t_i{}^{\ i\in\{1\ldots n\}-\{j\}}] \end{array}$$

The types $Obj(X.[l_i{:}\, T_i[X]^{i\in\{1\ldots n\}}])$ are used to describe objects, where X is a type quantifier representing the type of the object itself, and T_i is the type of the method named l_i relative to the object type X.

Our task is to provide an encoding of the three primitives and their types in the type theory. The type theory 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.

2.1 Record encoding

We interpret objects as records, where the record $\{l_1 = t_1, \ldots, l_n = t_n\}$ has fields $l_1, \ldots, l_n$ with values $t_1, \ldots, t_n$, 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 t. The record calculus is described in the following table:

For a record $r = \{l_i = t_i\}^{\ i\in\{1\ldots n\}}$:

$$\begin{array}{rcl} r \cdot l_j & \rightarrow & t_j \\ r \cdot l_j := t & \rightarrow & \{l_j = t, l_i = t_i{}^{\ i\in\{1\ldots n\}-\{j\}}\} \end{array}$$

The encoding of the object primitives in the record calculus is as follows:

$$\begin{array}{rcl} [l_i = \lambda x_i.t_i{}^{i\in\{1\ldots n\}}] & \equiv & \{l_i = \lambda x_i.t_i{}^{i\in\{1\ldots n\}}\} \\ o.l_j & \equiv & (o \cdot l_j)(o) \\ o.l_j \Leftarrow \lambda x.t & \equiv & o \cdot l_j := \lambda x.t \end{array}$$

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** is the equivalence class containing all terms (including the term **Top**).

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

3 Object Interpretation

Using the encoding of objects as functions, we would expect that the object types $Obj(X.[l_i{:}\, T_i{}^{\ i\in\{1\ldots n\}}])$ obey the equation

$$X = \{l_i{:}\, X \rightarrow T_i{}^{i\in\{1\ldots n\}}\}.$$

However, this type can't be expressed using general predicative recursive types because of the contravariant occurrence of X in the equation. For the moment, we restrict attention in this section to objects that have a well-founded call graph on the methods (objects with mutually recursive methods are considered in Section 3.3). To illustrate the properties of non-recursive objects, we fall back on the traditional example—the one-dimensional point, which has a field x that is an integer, and a method to produce a point shifted to the right by 1 unit.

$$\begin{array}{rcl} P & \equiv & Obj(X.[x{:}\, \mathbb{Z}; move{:}\, X]) \\ p_0 & \equiv & [x = \lambda o.0; move = \lambda o.(o.x \Leftarrow o.x + 1)] \end{array}$$

Although the coding

$$[\![P]\!] = \mu(X.\{x{:}\, X \rightarrow \mathbb{Z}; move{:}\, X \rightarrow X\})$$

is not well-formed, the x method does not use its argument, and the *move* method refers only to x. A more specific typing is:

$$[\![P]\!] = \{x{:}\, \{\} \rightarrow \mathbb{Z}; move{:}\, \{x{:}\, \{\} \rightarrow \mathbb{Z}\} \rightarrow \{x{:}\, \{\} \rightarrow \mathbb{Z}\}\}.$$

This type is not recursive, but it contains objects whose move method "forgets" about all methods but x. In order to get the desired behavior, the typing for *move* must specify that it does not affect any other method but x. In other words, it should be polymorphic over all object types that are extensions of $\{x{:}\, \{\} \rightarrow \mathbb{Z}\}$.

In the untyped calculus, polymorphism is expressed with the intersection type. The typing

$$move \in \bigcap\nolimits_k T \subseteq \{x{:}\, \{\} \rightarrow \mathbb{Z}\}.T \rightarrow T$$

specifies that the *move* method has the typing $T \rightarrow T$ for any T that is a subtype of $\{x{:}\, \{\} \rightarrow \mathbb{Z}\}$ in the predicative type universe $\mathbb{U}_k$. Unfortunately, this typing also fails to produce the expected behavior because it contains only the identity functions [24]. Essentially, we don't want polymorphism over *subtypes* of $\{x{:}\, \{\} \rightarrow \mathbb{Z}\}$, but over record *extensions* of $\{x{:}\, \{\} \rightarrow \mathbb{Z}\ \}$. Since the *move* method is to modify the value of the method x, the type for x must remain large enough to contain the modified value.

3.1 Constrained polymorphism

In order to address the subtyping problem, we must specify which methods are updatable. We adopt the method of *variance annotations* [2], where method labels are annotated with the symbols ($^+$, $^-$, $^\circ$), which are intended to provide the extra information. Object types are covariant in the types of their $^+$ methods, contravariant in the types of their $^-$ methods, and invariant in the type of their $^\circ$ methods (we do not consider the annotation $^-$).

Given the type annotations, we design a new constrained subtyping relation $\preceq$ to respect the type annotations. The subtyping relation $\subseteq$ retains its original semantics, and we have $A \preceq B \Rightarrow A \subseteq B$. Given an object type

$$T = \mathit{Obj}(X.[l_i \nu_i \colon T_i[X]^{i\in\{1\dots n\}}])$$

for $\nu_i \in \{^+, ^\circ\}$, let $\mathbf{L}_T^{upd}$ be the labels of the methods in T that are invariant. The set $\mathbf{L}_T^{upd}$ contains only the labels mentioned explicitly in T, and it is defined as follows:

$$\mathbf{L}_T^{upd} \equiv \{l_i \in \mathbf{Atom} \mid \nu_i = \ ^\circ\}.$$

Let the notation $T|_{\mathbf{L}}$ be the restriction of the object type T to the methods named in the set $\mathbf{L}$. We define the relation $\preceq$ as follows:

$$T' \preceq T \quad \textit{iff} \quad T' \subseteq T \wedge T'|_{\mathbf{L}_T^{upd}} \supseteq T|_{\mathbf{L}_T^{upd}}.$$

Essentially, we consider the types T' that are subtypes of T *and also* supertypes of T on the invariant methods (so T' and T are extensionally equivalent on the invariant methods). The intuition here is that the relevant subtypes must not further constrain the types of the invariant elements.

To make this definition more explicit, in the type theory we require that the interpretations of the types T' and T be dependent function types of the form $l\colon \mathbf{Atom} \to R_{T'}(l)$ and $l\colon \mathbf{Atom} \to R_T(l)$, for some types $R'_T(l)$ and $R_T(l)$. We restrict the subtyping condition to the type functions $R_{T'}$ and R_T to get the following definition:

$$\begin{array}{lll} T' \preceq T & \textit{iff} & \exists R_T, R'_T\colon \mathbf{Atom} \to \mathbf{Type}. \\ & & \quad [\![T]\!] = (l\colon \mathbf{Atom} \to R_T(l)) \\ & & \wedge\ [\![T']\!] = (l\colon \mathbf{Atom} \to R_{T'}(l)) \\ & & \wedge\ \forall l\colon \mathbf{Atom}.R_{T'}(l) \subseteq R_T(l) \\ & & \wedge\ \forall l\colon \mathbf{L}_T^{upd}.R_{T'}(l) \supseteq R_T(l) \end{array}$$

Note that this definition specifies that the methods of T' that do not appear in T are covariant relative to the object type T (there are *no* labels in $\mathbf{L}_T^{upd}$ that do not appear explicitly in the definition of T).

In the remainder of this paper, we will also use the notation $S' \preceq_T S$ to be the subobject relation on types relative to an object definition T. That is,

$$\begin{array}{lll} S' \preceq_T S & \textit{iff} & \exists R_S, R'_S\colon \mathbf{Atom} \to \mathbf{Type}. \\ & & \quad S = (l\colon \mathbf{Atom} \to R_S(l)) \\ & & \wedge\ S' = (l\colon \mathbf{Atom} \to R_{S'}(l)) \\ & & \wedge\ \forall l\colon \mathbf{Atom}.R_{S'}(l) \subseteq R_S(l) \\ & & \wedge\ \forall l\colon \mathbf{L}_T^{upd}.R_{S'}(l) \supseteq R_S(l) \end{array}$$

3.2 Non-recursive interpretation

Given this definition of restricted subtyping, the translation of object types for non-recursive objects can be given. Since the method call graph is acyclic, we arrange the methods in the object so that each method refers only to the previous methods, i.e. the method named l_i refers only to methods $l_1, \dots, l_{i-1}$.[1] We code the object type inductively for $T = \mathit{Obj}(X.[l_i \nu_i \colon T_i[X]^{i\in\{1\dots n\}}])$. Let T_i be the object type $\mathit{Obj}(X.[l_j \nu_j \colon T_j[X]^{j\in\{1\dots i-1\}}])$. The encoding for T is specified as follows:

$$\begin{array}{lll} [\![T_0]\!] & = & \{\} \\ [\![T_{i+1}]\!] & = & [\![T_i]\!] \cap \left\{ l_{i+1} \colon \bigcap_k T \preceq_{T_i} [\![T_i]\!] .T \to T_{i+1}[T/X] \right\}, \end{array}$$

where $[\![T]\!] \equiv [\![T_n]\!]$. Note that record intersection for records with distinct nontrivial fields is the same as record concatenation, and that $[\![T_{i+1}]\!] \preceq_{T_i} [\![T_i]\!]$ follows from the properties of record extension.

The issue of predicativity shows itself in the behavior of method selection. The rule we would like is the following:

$$\frac{\Gamma \vdash o \in T}{\Gamma \vdash o.l_i \in T_i[T/X]} \ \textbf{Obj sel}$$

For this rule to be valid, the method being selected must preserve the type T of the self parameter. Given the object $o \in T$, the full type of method l_i is

$$[\![o]\!]\,(l_i) \in \bigcap_k T' \preceq_{T_i} [\![T_{i-1}]\!] .T' \to T_i[T'/X].$$

By instantiating the quantifier with $[\![T]\!]$ itself, we would have

$$[\![o]\!]\,(l_i) \in [\![T]\!] \to T_i[[\![T]\!]\,/X],$$

and $[\![o]\!]\,(l_i)([\![o]\!]) \in T_i[[\![T]\!]\,/X]$, which is the desired result.

Unfortunately, even though $[\![T]\!] \preceq_{T_{i-1}} [\![T_{i-1}]\!]$, it is not possible to instantiate the quantification $\bigcap_k T' \preceq_{T_i} [\![T_{i-1}]\!] . \cdots$ directly, because of the occurrence of the type universe $\mathbb{U}_k$. This occurrence forces the intersection to belong to $\mathbb{U}_{k+1}$, which in turn forces the object type $[\![T]\!]$ into $\mathbb{U}_{k+1}$, prohibiting $[\![T]\!]$ from being used to instantiate the intersection. This is the crucial impredicative occurrence of self application. Before addressing this problem, we generalize the calculus to objects with mutually recursive methods.

3.3 Recursive Objects

Objects with recursive methods require additional interpretation because of the need to specify the recursion in the method types. To illustrate the issue, consider the following example of a point that contains a position and also a method to move the point to the factorial of its position. The factorial point has the following type:

$$P_F \equiv \mathit{Obj}(X.[x^\circ\colon \mathbb{N}; \mathit{fact}^+\colon X]).$$

An example of such a point is the following object:

$$p = \left[\begin{array}{l} x \quad\ = \lambda o.5; \\ \mathit{fact} = \lambda o.\,\mathbf{if}\ o.x = 0\ \mathbf{then} \\ \qquad\qquad\quad o.x \Leftarrow 1 \\ \qquad\qquad \mathbf{else} \\ \qquad\qquad\quad o.x \Leftarrow (o.x \Leftarrow o.x - 1).\mathit{fact}.x * o.x \end{array}\right]$$

The typing of this object is problematic because the *fact* method uses itself, and it needs its own type in order to derive a type. A non-recursive interpretation can be derived

[1] This ordering is done purely for exposition—it does not suppose that there is a single well-ordering ordering on the labels for all objects. Objects are considered equivalent if methods are reordered.

$$\begin{array}{lrcl}
\text{Let} & z & \equiv & [l_i = \lambda x_i.t_i \ {}^{i\in\{1\ldots n\}}] \\
& Z & \equiv & Obj(X.[l_i\nu_i{:}\, T_i[X] \ {}^{i\in\{1\ldots n\}}]) \\
& Z' & \equiv & Obj(X.[l_i\nu_i{:}\, T_i'[X] \ {}^{i\in\{1\ldots n+m\}}])
\end{array}$$

$$\frac{\Gamma, X_1, X_2{:}\,\mathbf{Type}, X_1 \subseteq X_2 \vdash T_i[X_1/X] \subseteq T_i[X_2/X] \quad \text{for } i \in \{1\ldots n\}}{\Gamma \vdash Z \in \mathbf{Type}} \ \textbf{Obj wf}$$

$$\frac{\Gamma \vdash Z \in \mathbf{Type} \quad \Gamma, T \preceq [l_i\nu_i{:}\, T_i[T/X] \ {}^{i\in\{1\ldots n\}}], o{:}\,T \vdash t_i[o/x_i] \in T_i[T/X]}{\Gamma \vdash z \in Z} \ \textbf{Obj mem}$$

$$\frac{\Gamma \vdash o \in Z}{\Gamma \vdash o.l_i \in T_i[Z/X]} \ \textbf{Obj sel}$$

$$\frac{\Gamma \vdash Z \in \mathbf{Type} \quad \Gamma, T \preceq [l_i\nu_i{:}\, T_i[T/X] \ {}^{i\in\{1\ldots n\}}], o{:}\,T \vdash t[o/x] \in T_i[T/X]}{\Gamma \vdash o.l_i \Leftarrow \lambda x.t \in Z} \ \textbf{Obj ovr}$$

$$\frac{\Gamma \vdash Z \in \mathbf{Type} \quad \Gamma \vdash Z' \in \mathbf{Type} \quad \Gamma \vdash T_i'[Z']^{\nu_i'} \lhd T_i[Z]^{\nu_i} \ \text{for } i \in \{1\ldots n\}}{\Gamma \vdash Z' \preceq Z} \ \textbf{Obj sub}$$

$$\frac{\Gamma \vdash S = T \in \mathbf{Type}}{\Gamma \vdash S^\circ \lhd T^\circ} \ \textbf{Sub Inv} \qquad \frac{\Gamma \vdash S \preceq T \ \text{for } \nu \in \{^\circ, ^+\}}{\Gamma \vdash S^\nu \lhd T^+} \ \textbf{Sub Cov}$$

Figure 2: Rules for nondependent calculus

from the following observation: method polymorphism is *class-based.* Once an object is instantiated, method use is monomorphic, with the type fixed at the type of the object. Methods are polymorphic, but recursive calls are *monomorphic.*

In the factorial point example, the recursive use of *fact* is at the type of the argument o, and the *fact* method can be given the following type:

$$fact{:}\bigcap_k T \preceq_{P_F} \{x{:}\,\{\} \to \mathbb{Z}; fact{:}\,T \to T\}.T \to T.$$

The interpretation of the object type T is *partial* to account for possible divergence of the *fact* method. Until now, the interpretation has not distinguished between partial and total objects. However, with the introduction of recursive methods, a formal account of totality would require a specification of all the intermediate computations. Our account of partial objects is based on the partial types of Smith [25], which have been simplified by Crary [7]. A partial type, denoted $\overline{T}$, contains a term x if the evaluation of x diverges, or if x is an element of T. The induction principle for partial types relies on an *admissiblity* condition, which holds for the types in our calculus. In the rest of the section, we interpret all types as partial.

The interpretation of objects is given in terms of its monomorphic and polymorphic translations. The monomorphic interpretation is used for the object specification in recursive calls. For an object $T = Obj(X.[l_i\nu_i{:}\, T_i[X] \ {}^{i\in\{1\ldots n\}}])$ the interpretation is given with respect to an object type O:

$$[\![T]\!]^O_{mono} \equiv \{l_i{:}\, O \to T_i[O/X] \ {}^{i\in\{1\ldots n\}}\}.$$

The general interpretation requires that methods also be polymorphic over all subobjects of the monomorphic interpretation.

$$[\![T]\!] \equiv \left\{l_i{:}\bigcap_k O \preceq_T [\![T]\!]^O_{mono}.O \to T_i[O/X] \ {}^{i\in\{1\ldots n\}}\right\}$$

For example, when applied to the point example P_F, these definitions give the following interpretations:

$$[\![P_F]\!]^O_{mono} \equiv \{x{:}\, O \to \mathbb{Z}; fact{:}\, O \to O\}$$

and

$$[\![P_F]\!] \equiv \left\{ \begin{array}{l} x{:}\bigcap_k O \preceq_{P_F} [\![P_F]\!]^O_{mono}.O \to \mathbb{Z}; \\ move{:}\bigcap_k O \preceq_{P_F} [\![P_F]\!]^O_{mono}.O \to O \end{array} \right\}.$$

To complete the example, the relation $O \preceq_{P_F} [\![P_F]\!]^O_{mono}$ has the following definition:

$$\begin{array}{l}
O \preceq_{P_F} [\![P_F]\!]^O_{mono} \equiv \\
\quad \exists R{:}\, \mathbf{Atom} \to \mathbf{Type}. \\
\quad\quad O = l{:}\, \mathbf{Atom} \to R(l) \\
\quad \wedge\ R(x) \subseteq O \to \mathbb{Z} \wedge R(x) \supseteq O \to \mathbb{Z} \\
\quad \wedge\ R(fact) \subseteq O \to O
\end{array}$$

The types in the the point definitions are considered to be partial, to allow for points with nonterminating methods, and the relation $\preceq_{P_F}$ constrains the polymorphism to subobjects that are invariant in their x method (to allow for updating), and covariant in their *fact* method.

Using the definitions for the general object interpretations, the rules of the object calculus can be derived from the rules of the type theory. The rules for the nondependent object calculus are listed in Figure 2. These rules justify the “standard” rules given in Abadi et. al. for the object calculus. The relation $\lhd$ is defined to simplify the rules for subtyping, and is defined in the type theory as follows:

$$\nu_1 T_1 \lhd \nu_2 T_2 \equiv \begin{cases} T_1 = T_2 \in \mathbf{Type} & \text{if } \nu_1 = ^\circ, \nu_2 = ^\circ \\ T_1 \preceq T_2 & \text{if } \nu_1 \in \{^\circ, ^+\}, \nu_2 = ^+ \\ \mathbf{False} & \text{otherwise} \end{cases}$$

The soundness of these rules have been verified in the Nuprl theorem prover. The proof relies critically on method monotonicity, which specifies that condition under which self-application is allowed.

Definition 1 *Monotonicity. A family of types $\mathcal{T}\colon \mathbf{Type} \to \mathbf{Type}$ is* monotonic *if for any two types T_1 and T_2 where T_1 is a subtype of T_2, then $\mathcal{T}(T_1)$ is a subtype of $\mathcal{T}(T_2)$. Equivalently:*

$$\forall T_1, T_2\colon \mathbf{Type}. T_1 \subseteq T_2 \Rightarrow \mathcal{T}(T_1) \subseteq \mathcal{T}(T_2)$$

For the interpretation of objects, we use monotonicity to infer continuity over the intersection type. For the type theory fragment we use, all types are continuous if they are monotone.

Definition 2 *Continuity. A family of types $\mathcal{T}\colon \mathbf{Type} \to \mathbf{Type}$ is* continuous *relative to a type T if the following relation holds:*

$$\left(\bigcap T' \subseteq T. \mathcal{T}(T')\right) = \mathcal{T}\left(\bigcap T' \subseteq T. T'\right)$$

Theorem 1 *Soundness. The rules for non-dependent objects are sound with respect to the type theoretic model.*

As we mentioned in Section 3.2, the critical rule in the verification is the rule for **Obj sel**, which requires a self-application. We outline the critical step of the proof here, which is the typing of the self-application for $o \in [\![Z]\!]$:

$$[\![o]\!]\,(l_i)([\![o]\!]) \in T_i[[\![Z]\!]\,/X]$$

The interpretation $[\![Z]\!]$ is a record of polymorphic functions. Since the record is monotonic in the types, we can normalize $[\![Z]\!]$ so that all intersections occur outside the record:

$$[\![Z]\!] = \bigcap\nolimits_k O \preceq_Z [\![Z]\!]^O_{mono}.\{l_i\colon O \to T_i[O/X]^{\ i \in \{1\ldots n\}}\}$$

To simplify the presentation, we make the following definitions:

$$\begin{array}{rcl} O' & \equiv & [\![Z]\!]^O_{mono} \\ \mathcal{T} & \equiv & \lambda O.\{l_i\colon O \to T_i[O/X]^{\ i \in \{1\ldots n\}}\}, \end{array}$$

and

$$[\![Z]\!] = \bigcap\nolimits_k O \preceq_Z O'.\mathcal{T}(O).$$

Given the continuity of T_i, the statement

$$[\![o]\!]\,(l_i)([\![o]\!]) \in T_i\left[\bigcap\nolimits_k O \preceq_Z O'.\mathcal{T}(O)./X\right]$$

can be rewritten to the equivalent statement

$$[\![o]\!]\,(l_i)([\![o]\!]) \in \bigcap\nolimits_k O \preceq_Z O'.T_i\left[\mathcal{T}(O)/X\right]$$

which is provable using a direct case analysis.

Although there are several technical steps in this argument, the crucial point is semantic: method monotonicity preserves the predicativity of type intersections. In turn, the type intersections are crucial for describing polymorphism, which provides the basis for two fundamental features of object-oriented languages: method subsumption and inheritance. An extension of this interpretation to binary methods would need to answer the following question: is there a predicative interpretation of type intersections of non-monotonic type families?

While the semantic argument is useful for understanding the mathematics of inheritance, the proof-theory shows how monotonicity is used to restrict proofs to cases with well-known behavior. We give an additional proof-theoretic argument for **Obj sel** in Appendix **??**.

Several results follow directly from the interpretation. Our interpretation of objects in the type type theory inherits properties from each of its models. Progress, preservation, and substitution properties, which normally require proof for object systems, arise directly from a semantics of *functionality* for the type theory. In addition, the type theoretic interpretation wraps the object system in a mathematical framework, making it suitable as a basis for inheritance of *proofs*, the subject of the next section.

4 Dependent Objects

The interpretation of objects can also be extended with *dependent* object types, which can be used for defining classes that contain formal specifications (and objects with *proofs*). There are two main issues that arise in the interpretation:

1. Methods form groups in which all methods must be updated simultaneously to preserve the typing properties.

2. The interpretation of types relies on *dependent* record types, which are defined as *very*-dependent function types.

Dependent method types use Curry-Howard isomorphism to assert propositions about the implementation, and the method implementations are proofs that the specification is satisfied.

In the dependent object calculus, we require that the type dependencies in object types be well-ordered. For object types, we keep the familiar notation

$$Obj(X.[l_i = T_i[X]^{\ i \in \{1\ldots n\}}]),$$

and we allow the types $T_i[X]$ to contain occurrences of the labels $l_1, \ldots, l_{i-1}$, which represent the values of the corresponding methods. For instance, the quantification $\exists x\colon A.B_x$ is isomorphic to the object type $Obj(X.[x\colon A; p\colon B_x])$.

In this section, we extend the P example to two dimensional points on the unit circle:

$$P_{2D} \equiv Obj(X.[x, y\colon \mathbb{R}; p\colon x^2 + y^2 \approx 1; rot : \mathbb{R} \to X])$$

The p method is a *proposition* that constraints the coordinates x and y to lie on the unit circle. The *rot* method is intended to rotate the point by a given angle. Note that the value for x can't normally be updated without also updating the value for y. In general, methods form groups (in this case $\{x, y\}$) in which all of the methods must be updated simultaneously to preserve the specification.

Since methods are required to update members of dependency groups simultaneously, it is reasonable to constrain the polymorphism of methods directly. Rather than using variance annotations on the object methods, we shift the annotations to the polymorphic quantification within the method types. For example, for the P_{2D}, we use the typing

$$rot\colon \bigcap\nolimits_k T \preceq_d Obj(X.[x^\circ, y^\circ\colon \mathbb{R}; p^+\colon x^2{+}y^2 \approx 1]).T \to \mathbb{R} \to T.$$

This specification uses an even more constrained subtyping relation $\preceq_d$ that we discuss shortly, and the quantification is over all the methods and constraints that are to be updated.

$$
\begin{array}{llll}
\text{Let} & z \equiv [l_i = \lambda x_i.T_i \ ^{i\in\{1\ldots n\}}] & & \\
& Z \equiv Obj(X.[l_i\nu_i\colon T_i[X] \ ^{i\in\{1\ldots n\}}]) & Z_i \equiv & Obj(X.[l_j\nu_j\colon T_j[X] \ ^{j\in\{1\ldots i\}}]) \\
& Z' \equiv Obj(X.[l_i\nu_i\colon T'_i[X] \ ^{i\in\{1\ldots n+m\}}]) & Z'_i \equiv & Obj(X.[l_j\nu_j\colon T'_j[X] \ ^{j\in\{1\ldots i\}}])
\end{array}
$$

$$
\frac{\Gamma, o\colon Z_{i-1}, X_1, X_2\colon \mathbf{Type}, X_1 \subseteq X_2 \vdash T_i[o/\bar{l}, X_1/X] \subseteq T_i[o/\bar{l}, X_2/X] \quad \text{for } i \in \{1\ldots n\}}{\Gamma \vdash Z \in \mathbf{Type}} \ \textbf{Obj wf}
$$

$$
\frac{\Gamma \vdash Z \in \mathbf{Type} \quad \Gamma, T \preceq Z_{i-1}, o\colon T \vdash t_i[o/x_i] \in T_i[o/\bar{l}, T/X]}{\Gamma \vdash z \in Z} \ \textbf{Obj mem}
$$

$$
\frac{\Gamma \vdash o \in Z}{\Gamma \vdash o.l_i \in T_i[\bar{l}, Z/X]} \ \textbf{Obj sel}
$$

$$
\frac{\Gamma \vdash Z \in \mathbf{Type} \quad \Gamma, T \preceq Z_{i-1}, o\colon T \vdash t[o/x] \in T_i[o/\bar{l}, T/X]}{\Gamma \vdash o.l_i \Leftarrow \lambda x.t \in Z} \ \textbf{Obj ovr}
$$

$$
\frac{\begin{array}{l}\Gamma \vdash Z \in \mathbf{Type} \\ \Gamma \vdash Z' \in \mathbf{Type} \\ \Gamma, o\colon Z'_{i-1} \vdash T'_i[o/\bar{l}, Z'_{i-1}/X]^{\nu'_i} \lhd T_i[o/\bar{l}, Z_{i-1}/X]^{\nu_i} \ \text{for } i \in \{1\ldots n\}\end{array}}{\Gamma \vdash Z' \preceq Z} \ \textbf{Obj sub}
$$

$$
\frac{\Gamma \vdash S = T \in \mathbf{Type}}{\Gamma \vdash S^\circ \lhd T^\circ} \ \textbf{Sub Inv} \qquad \frac{\Gamma \vdash S \preceq T \ \text{for } \nu \in \{^\circ, ^+\}}{\Gamma \vdash S^\nu \lhd T^+} \ \textbf{Sub Cov}
$$

Figure 3: Rules for dependent calculus

4.1 Encoding

We encode dependent object types as *dependent* records, which in turn are encoded as *very-dependent* function types [11]. Very-dependent function types allow function values to be used to specify their range, in a well-ordered manner. We use the notation

$$\{f \mid x\colon A \to B[f, x]\}$$

to specify the functions f with domain type A, and range type $B[f, a]$ on argument $a \in A$. The domain A must be well-ordered, and $B[f, a]$ may call f only on values $a' \in A$ that are *smaller* than a. For objects and records, the domain type A is the type of labels **Atom**. We always write objects so that the label order is the well-order we are using over the domain **Atom** (smaller elements first).

The encoding of dependent records as very-dependent functions is as follows. Given a dependent record type $R = \{l_i\colon T_i \ ^{i\in\{1\ldots n\}}\}$, where the labels $l_1, \ldots, l_{i-1}$ may occur in the types T_i, we have the encoding:

$$
R \equiv \left\{ \begin{array}{ll} f \mid l\colon \mathbf{Atom}.\ \mathbf{case}\, l\, \mathbf{of} & \\ & \mid \cdots \\ & \mid l_i\colon T_i[f(l_j)/l_j]^{j\in\{1\ldots i-1\}} \\ & \mid \cdots \\ & \mid \_\colon \mathbf{Top} \end{array} \right\}
$$

We modify object types so that the methods specify the method groups they are selecting and updating adjacent to their method label. The general form is:

$$Obj(X.[l_i(\overline{a_i})\colon B_i[X] \ ^{i\in\{1\ldots n\}}]),$$

where $\overline{a_i}$ is a vector of annotations $l_j\colon \nu_j$. We typically leave out the annotations if the vector $\overline{a_i}$ is empty. For instance, the fully annotated P_{2D} type is

$$P_{2D} \equiv Obj\left(X.\left[\begin{array}{l} x, y\colon \mathbb{R}; p\colon x^2 + y^2 \approx 1; \\ rot(x^\circ, y^\circ, p^+)\colon \mathbb{R} \to X \end{array}\right]\right),$$

which states that the *rot* method updates x and y, and it refers to p (perhaps implicitly). Note that the type

$$P_{2D} \equiv Obj(X.[x, y\colon \mathbb{R}; p\colon x^2 + y^2 \approx 1; rot(x^\circ, p^+)\colon \mathbb{R} \to X])$$

is a valid type, but it does not permit the *rot* method to update the value of y.

The encoding of object types is similar to the nondependent case. Consider an object type

$$O \equiv Obj(X.[l_i(\overline{a_i})\colon T_i[X] \ ^{i\in\{1\ldots n\}}])$$

that does not contain recursive method, where

$$\overline{a_i} \equiv a_{i_1}\colon \nu_{i_1}, \ldots, a_{i_m}\colon \nu_{i_m}.$$

The encoding for O is defined as follows:

$$
\begin{array}{rcl}
[\![T_0]\!] & = & \{\} \\
[\![T_{i+1}]\!] & = & [\![T_i]\!] \cap \left\{ l_{i+1}\colon \bigcap_k T \preceq_d [\![T_i]\!].T \to T_{i+1}[T/X] \right\}
\end{array}
$$

and $[\![O]\!] = [\![T_n]\!]$.

4.2 Dependent subtyping: $\preceq^{dep}$

The dependent encoding is essentially the same as the non-dependent case, except for the occurrence of the $\preceq_d$ relation, and the fact that each method has its own variance annotations. The $\preceq_d$ relation is intended to prevent dependencies from adversely constraining invariant types. For example, the type $[x^\circ{:}\mathbb{Z}; p{:}x = 0 \in \mathbb{Z}]$ constrains the type for x, even though it has the same method type for x as the object type $[x^\circ{:}\mathbb{Z}]$. The $\preceq_d$ prevents this scenario by explicitly stating that invariant elements are overridable. Let $T \equiv Obj(X.[l_i \nu_i{:}T_i[X]\ ^{i \in \{1 \ldots m\}}])$, and let *upd* be the collection of indices of invariant methods in T, and let L_T^{upd} be the corresponding collection of labels.

$$
\begin{aligned}
T' \preceq_d T \equiv\ & T' \subseteq T \\
& \wedge \forall t{:}T'.\forall z{:}\ Obj(X.[l_i{:}T_i\ ^{i \in upd}]).\exists t'{:}T. \\
& \qquad t' = t \in \mathbf{Atom}/L_T^{upd} \to \mathbf{Type} \\
& \qquad \wedge\ t' = z \in L_T^{upd} \to \mathbf{Type}
\end{aligned}
$$

This definition states explicitly that the subtypes under consideration are only those where values for the invariant methods can be updated with any values that are mutually consistent. Given this definition, we can derive the rules shown in Figure 3 for the dependent object calculus. In the table, we use the notation $T_i[o/\bar{l}]$ to mean the substitution $T_i[o.l/l]$ for each $l \in \bar{l}$ where $\bar{l}$ is the set of all labels in T_i.

Theorem 2 *The rules for dependent objects are sound.*

The proof of soundness is essentially the same as for the nondependent-rules, except to account for the additional bindings in the method types.

As before, the type theory justifies standard properties for the object calculus. The functionality of the type theory guarantees progress, preservation, and substitution properties.

5 Related Work

Another framework for expressive object calculi is developed by Hofmann et el. [13], who have implemented a verification calculus based on the existential interpretation of objects. In their system, objects have three parts: an object state, the object methods, and the proofs of method correctness (dependent objects with exactly one dependency). Their interpretation uses an existential encoding of objects [21, 16] in an impredicative type theory. Although the existential encoding restricts method update, the interpretation of objects does not require the use of recursive types, allowing the development of expressive type systems [6, 14].

The interpretation of object relies on a restricted subtyping relation (we use notation $\preceq$) to describe valid subobjects. This is important because the general subtype relation would disallow updates. Our restricted subtyping is essentially the same as the *positive subtyping* of Hofmann and Pierce [14]. In the definition of restricted subtyping, we describe which parts of the type are not allowed to change. Hofmann and Pierce describe it in terms of coercions: $S \preceq T$ if there is a coercion $S \to T$, and there is a function in $S \to T \to S$ that replaces the "T" part of S with a new element of T. In their account, Hofmann and Pierce use the subtyping to define an existential encoding of objects in $F^\omega_\leq$..

Another existential system was developed by Jackson [18] to formalize a significant portion of constructive algebra in Nuprl. While Jackson's framework includes many of the properties of objects, the subtyping requirements prohibit the use of self-application and constrain the shape of objects. Our work has borrowed from Jackson' interpretation; in an earlier paper [11], we presented another existential interpretation of objects in a predicative type theory, using the very dependent function type to express the type dependencies.

6 Conclusion

We have given a interpretation of objects that differs from more standard accounts by replacing the use of recursive types for objects with polymorphic restrictions on methods. The elimination of recursive types enables a predicative interpretation that provides new mathematical models of objects. The design of object-oriented languages is notoriously difficult, and mathematical insight aids in the development of languages that are clean, powerful, correct, and elegant.

There are several areas for future work, including interpretations of binary methods, which occur in practice. In our interpretation, objects are coerced to monomorphic type within method bodies, which may limit the use of dynamic inheritance in languages that add new methods to objects at run time. It may be possible to address this issue by giving a syntactic account of polymorphism that allows the use of recursive types to define objects. We are using the interpretation in the Nuprl-Light system [12] to implement a formal module layer on top of the Objective Caml module system [22].

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[2] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An Interpretation of Objects and Object Types. In *ACM Symposium on Principles of Programming Languages*, 1996.

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

[4] Stuart F. Allen. A non-type-theoretic definition of Martin-Lof's types. In *Proceedings of the Second Conference on Logic in Computer Science*, pages 215–224, June 1987.

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

[6] Adriana B. Compagnoni and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 1996.

[7] Karl Crary. Recursive computation in foundational type theory. Technical report, Department of Computer Science, Cornell University, Forthcoming.

[8] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Logic in Computer Science,* 1993, 26–38.

[9] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'a rithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

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

[11] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page at Williams College.

[12] Jason J. Hickey. Nuprl-Light: An implementation framework for higher–order logics. In *14th International Conference on Automated Deduction*. Springer, 1997.

[13] M. Hofmann, W. Naraschewski, M. Steffen, and T. Stroup. Inheritance of proofs. In *TAPOS*. Wiley, forthcoming.

[14] M. Hofmann and B.C. Pierce. Positive subtyping. *Information and Computation*, 126(1):11–33, 1996. Preliminary version appeared in *Principles of Programming Languages*, 1995.

[15] Martin Hofmann, Wolfgang Narashewski, Martin Steffan, and Terry Stroup. Inheritance of proofs. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page at Williams College.

[16] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995.

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

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

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

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

[21] 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.

[22] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.

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

[24] Robinson and Tennent. Bounded quantification and record-update problems. Message to `types` email list, 1988.

[25] Scott Fraser Smith. *Partial Objects in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

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.

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

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:

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

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 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 $k \in \omega$. 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 partial type $\overline{A}$ contains the elements of A, and also all divergent computations.