A Modest Model of Records, Inheritance, and Bounded Quantification*

KIM B. BRUCE†

Computer Science Department, Williams College,
Williamstown, Massachusetts 01267

AND

GIUSEPPE LONGO‡

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, I-56125 Pisa, Italy

1. INTRODUCTION

Over the last several years there has been growing interest in object-oriented languages in the programming languages community, and this interest has spread to other technical communities such as those in software engineering and data base. (Note. In this paper we consider object-oriented languages to be those which include support for inheritance on subtypes.) Simula 67 (Birtwistle et al., 1973) and Smalltalk (Goldberg and Robson, 1983) in their various incarnations represent the earliest work on object-oriented languages. More recently other languages have been developed including C++ (Stroustrup, 1986) and VBase (Andrews and Harris, 1986) (both based on C), Flavors (Weinreb and Moon, 1981) and LOOPS (Bobrow and Stefik, 1982) (extensions of LISP), OWL (Schaffert et al., 1986) (based on Clu), and Eiffel (Meyer, 1988).

Cardelli (1988) (an earlier version of which appeared in 1984) developed one of the earliest formal approaches to inheritance, introducing both the syntax and semantics of an extension of the classical typed lambda calculus supporting inheritance. Building on some earlier work on type containment

* An earlier version of this paper appeared in the Proceedings of the Third Annual Symposium on Logic in Computer Science, 1988. Partially supported by NATO Civil Alliance Grant # RG.86/0129 for international collaboration in research, "The semantics of types in programming languages."
† Partially supported by NSF Grant DCR-8603890 and a grant from Williams College.
‡ This author's work has been made possible also by the generous hospitality of the Computer Science Dept., Carnegie-Mellon University, while teaching there during the academic year 1987/88.
by Mitchell (1988), Cardelli and Wegner (1985) (from now on CW, 1985) presented an extension of the second-order lambda calculus supporting inheritance. This language supported both parametric and subtype (inheritance) polymorphism by means of bounded quantification. We will use the notation introduced by CW (1985), including their name for the language, Bounded Fun.

The formal definition of subtype, based on the notion of inheritance, is based on the intuition that if a function may be applied to an argument of type $\tau$ then it should make sense to apply it (in some natural way) to an argument of type $\sigma$, for $\sigma \subseteq \tau$. In particular, suppose that $\sigma$ and $\tau$ are record types and that $\sigma$ contains all of the same fields as $\tau$ (each with the same type as the corresponding field in $\tau$), with possibly some extra fields. Then $\sigma \subseteq \tau$, since any function which can be applied to an argument of type $\tau$ only depends on the fact that the argument has particular fields which appear in $\tau$. Having extra fields causes no difficulty, so conceptually it makes sense to apply this function to elements of type $\sigma$ as well. The point now is to give a precise mathematical meaning to this, in particular in the context of higher order languages.

In Bounded Fun, we use $\text{fun}(x : \sigma). e$ to denote a function which takes an argument of type $\sigma$, $\text{all } t. e$ to denote a function which takes a type parameter, and $\text{all } t \leq \tau. e$ to denote a function which takes a type parameter which is restricted to be a subtype of $\tau$. The following example will give the flavor of the language.

Let $\tau$ be a record type with (at least) fields "elts" and "<", where "elts" has type "Array of $t$" and "<" has type $t \times t \rightarrow \text{bool}$, for some type $t$. Let $\text{fun}(u : z). e$ be a term which, when given input $u : z$, sorts the array in the field "elts" and leaves the output unchanged in the other fields. This program may work on all "sortable" records; that is, on all records which contain (at least) the fields "elts" and "<". Formally, let

$$\text{sortablerec}(t) = (\text{elts} : \text{Array of } t, < : t \times t \rightarrow \text{bool}).$$

Then $\text{fun}(u : \tau). e$ would work on any $u : \tau$ provided that $\tau \leq \text{sortablerec}(t)$, for some $t$. The following polymorphic function will perform a sort on any "sortable" record:

$$\text{sort} = \text{all } t. \text{all } s \leq \text{sortablerec}(t). \text{fun}(u : s). e.$$

If we wish to apply sort to a record $u_0$ of record type

$$\sigma = (\text{elts} : \text{Array of } \text{Int}, < : \text{Int} \times \text{Int} \rightarrow \text{Bool}, ...),$$

we simply write $\text{sort}[\text{Int}][\sigma](u_0)$. Thus one can imagine writing quite powerful and expressive programs in this extension of the typed \lambda-calculus.
(although, see Section 6), where functions can be applied to elements, to
types, and even to restricted collections of types (see CW, 1985, for further
examples). Languages in which these type parameters may be omitted may
be understood as abbreviations of Bounded Fun. Thus the results in this
paper can be seen to be quite general.

Mitchell (1988) was primarily concerned with type inference in a similar
language, while (CW, 1985) is an exposition of various extensions of
the lambda calculus, including those supporting polymorphism and
inheritance. In both Cardelli (1988) and CW (1985), the authors suggest
that the semantics of these languages may be understood in terms of the
ideal model developed by MacQueen, Plotkin, and Sethi (1984). In these
models, types are interpreted as ideals in a cpo (complete partial order)
which is rich enough to model all of the data types of the model. These
ideal models were originally developed to model languages like ML. Ideal
models are models of the untyped lambda calculus with an associated type
inference scheme to infer possible types for terms. Unfortunately these
models are not sound for the typed lambda calculus (and extensions)
because of the failure of weak extensionality (see Bruce et al., 1988 for
further discussion). Also, they do not give meaning to either polymorphic
application (applying a term to a type) or second-order terms.

This paper originated in an attempt to formulate a sound model for
Bounded Fun. Bruce and Wegner (1987) present an abstract model of
inheritance based on algebraic models. Our definition of a model for
Bounded Fun integrates those ideas with the model definition for the
second-order lambda calculus given in Bruce et al. (1990). In particular,
this new definition models the notion of subtype by the existence of a
"natural coercer" from the type to a supertype. Thus if an expression, e, of
type $A < B$ is used in a context where an element of type $B$ is expected, the
meaning of $e$ in type $A$ can be coerced to an element of type $B$. Given the
general definition, the principal aim of this paper is to provide a concrete
model for Bounded Fun. This model is based on the per (partial equiv-
amence relation) model for the second-order lambda calculus. (Note. The
sets involved in these models are sometimes called Modest sets and are the
objects of the category $\mathbf{M}$ below.) These models have a long and complex
history involving successive inventions (and reinventions) of concepts by
Troëlstra (1972), Girard (1972), and most recently by Moggi (1986), and
subsequently by Mitchell (1986), Breazu-Tannen and Coquand (1987), and
others. More recent work investigating the connection of this work to
category theory and intuitionism includes Hyland (1988), Hyland et al.
(1990), Hyland and Pitts (1987), Carboni et al. (1987), Pitts (1987), Longo

In this new model, types are interpreted as partial equivalence relations
(relations which are symmetric and transitive, but not necessarily reflexive)
over \( \omega \). The set of elements of a type, \( A \), is the set of equivalence classes of \( A \), \( \mathcal{Q}(A) = \{ \{ n \}_A \mid n \in A \} \), where \( \{ n \}_A = \{ m \mid m \in A \} \).

In object-oriented programming languages subtypes usually arise in two principal ways:

(i) The elements of a subtype form a subset of the supertype.

(ii) In record types, the subtype has all of the fields of the supertype with possibly more fields.

Note that when these are combined, more complex combinations are possible.

These two aspects of subtypes are captured easily by partial equivalence relations (pers). If fewer elements are desired, simply throw away some equivalence classes (it is a \emph{partial} equivalence relation, after all). In case (ii), the argument is a bit more subtle. If \( A \preceq B \) because it has more fields, then \( A \) can make finer distinctions than \( B \) (since element of \( A \) contain more information than elements of \( B \)). This can be modeled by making the per for \( A \) a refinement of \( B \). Thus if \( m \) and \( n \) are related according to \( A \), they must be related according to \( B \). Conversely, if \( m \) is distinguishable from \( n \) in \( B \), then it is still distinguishable in \( A \). Nevertheless, we may have \( m \) unrelated to \( n \) via \( A \), but related according to the more limited information available to \( B \).

Amazingly, both of these notions (of throwing away equivalence classes and taking refinements of partial equivalence relations) correspond to the per of the subtype being a subset of the per of the supertype. That is, we will define \( A \preceq B \iff A \subseteq B \) (when each is looked at as a set of ordered pairs). Note, however, that since the elements of types \( A \) and \( B \) are equivalence classes, it will typically not be the case that the set of elements of type \( A \) is a subset of the set of elements of type \( B \). In particular, this will fail when \( A \) is a proper refinement of \( B \). The "natural coercer" from \( A \) to \( B \) referred to earlier is the natural map which takes the element \( \{ n \}_A \) of type \( A \) to \( \{ n \}_B \) of type \( B \), where \( \{ n \}_A \) is the equivalence class of \( n \) in \( A \).

In the earlier presentations of the per models, polymorphic types were presented as intersections of their instantiations. E.g., \( \forall t. t \rightarrow t \eta = \bigcap_{A \in \text{Type}} (A \rightarrow A) \). Following the category theoretic approach mentioned above, a more natural interpretation as "indexed products" may be given, which turns out to be isomorphic to the intersection interpretation. In terms of the previous example, one has

\[
\forall t. t \rightarrow t \eta = \left[ \prod_{A \in \text{Type}} (A \rightarrow A) \right]_M \cong \bigcap_{A \in \text{Type}} (A \rightarrow A),
\]

for \( \prod_{A \in \text{Type}} (A \rightarrow A) \) as defined in Section 5 (see Longo and Moggi, 1988, for details and category theoretic justifications). Also records are
interpreted by "indexed products," since they may be viewed as dependent types (Section 3). By this, the approach we propose unifies the mathematical understanding of polymorphic, dependent, and record types.

The key point with the semantics below is its simplicity and its "set-theoretic" flavor. Indeed, we can give a very simple interpretation of records and inheritance for a polymorphic language, since we (implicitly) work in (a model of) intuitionistic set theory. The model, Hyland's Effective Topos, is hidden in the background of our elementary treatment but gives it structural significance and the ultimate motivation (see Pitts, 1987; Longo and Moggi, 1988).

The paper begins with a brief introduction to the typed lambda calculus with records and subtypes in Section 2. This language is a slight simplification of that given in Cardelli (1988). In Section 3 we show informally how to construct a model of this language from partial equivalence relations. It is our hope that this informal introduction will provide the reader with a solid intuition on which to base the study of the more complex language, Bounded Fun.

In Section 4 we begin by presenting the syntax and semantics of Minimal Bounded Fun. Minimal Bounded Fun is essentially a weakening of the usual Bounded Fun by dropping the "subsumption" rule from the type inference system. We then introduce an extension of the language obtained by adding a constant, convert, and appropriate rules to ensure that this constant is interpreted as a well-behaved coercion function. Technical results about terms with the same "erasures" in this language allow us to use models of this richer language to interpret Bounded Fun. In Section 5 we introduce \( \omega \)-sets and show how to construct a model for Bounded Fun by using the \( \omega \)-sets to interpret kinds, and modest sets (pers) to model the types of the language. In Section 6 we discuss problems with Bounded Fun that have appeared as a result of examining this model and possible directions for future work to improve the language. Section 7 provides a summary of the paper and places it within the context of other recent work.

2. The Typed Lambda Calculus with Records and Subtypes

In this section we present an informal overview of the typed lambda calculus with records and subtypes. We presume that the reader is familiar with the syntax and semantics of the classical typed lambda calculus. As a brief reminder, we note that we will write fun \((x : \sigma). e\) for the function with body \(e\) and formal parameter \(x\) of type \(\sigma\). \((ee')\) will denote function application as usual. We write \(e : \tau\) to indicate that \(e\) has type \(\tau\). The material in this section is adapted from CW (1985).

We begin by introducing record types.
2.1. Definition. Let $L$ be a set of labels or identifiers.

(i) $(I_1 : A_1, \ldots, I_n : A_n)$ is a **record type** if $A_1, \ldots, A_n$ are types and $I_1, \ldots, I_n$ are in $L$;

(ii) $(I_1 = a_1, \ldots, I_n = a_n) : (I_1 : A_1, \ldots, I_n : A_n)$ if $a_1 : A_1, \ldots$, and $a_n : A_n$.

For example:

$$\text{car-type} = (\text{make : string, model : string, year : int})$$

is a record type and

$$\text{my-car} = (\text{make = Fiat, model = Panda, year = 1986})$$

is a record of type **car-type**.

Thus a record type is a finite set of associations of identifiers (corresponding to the fields of the record) with types. Its elements are functions from the set of (component) identifiers to types such that each identifier is sent to an element of the corresponding type. More formally,

$$f \in \{ (I_1 : A_1, \ldots, I_n : A_n) \} \eta \text{ iff for all } 1 \leq i \leq n, \ f(\{I_i\} \eta) \in \{A_i\} \eta.$$ 

For example, the interpretation $\llbracket \text{my-car} \rrbracket \eta$ of the record \text{my-car} above is the function, $f$, such that

$$f(\llbracket \text{make} \rrbracket \eta) = \llbracket \text{Fiat} \rrbracket \eta \in \llbracket \text{String} \rrbracket \eta,$$

$$f(\llbracket \text{model} \rrbracket \eta) = \llbracket \text{Panda} \rrbracket \eta \in \llbracket \text{String} \rrbracket \eta,$$

$$f(\llbracket \text{year} \rrbracket \eta) = \llbracket 1986 \rrbracket \eta \in \llbracket \text{Int} \rrbracket \eta.$$ 

As mentioned in the Introduction, subtypes are used in object-oriented programming languages such as Smalltalk, C++, Owl, etc., as a way of allowing subtypes to inherit operations from their supertypes (see Bruce and Wegner, 1987, for a more complete discussion of inheritance in terms of behavioral compatibility).

In this context, we will characterize $A \leq B$ by the existence of a "natural" coercion function which takes elements of $A$ to elements of $B$. Thus if $\text{coerce}_{A,B}$ is such a coercion function and $f : B \rightarrow C$, then $f$ can be applied to an element $a$ of $A$ by computing $f(\text{coerce}_{A,B}(a))$. In the modest model in this paper, $\text{coerce}_{A,B}$ will be a very natural (although not necessarily injective) function (see Section 3 and Remark 5.2.2). Depending on the language design, one can make these coercion functions part of the language (requiring them to appear explicitly in computations) or let a type-checker infer them where necessary. Whichever way is chosen, one may use the system given below to infer type inclusions.
2.2. DEFINITION. An inequality of the form $\sigma \leq \tau$, where $\sigma$, $\tau$ are type expressions, is said to be a type constraint. If, moreover, $t$ is a type variable then we say $t \leq \tau$ is a simple type constraint which declares $t$. If $t \leq \tau$ is included in a set $C$ of simple type constraints then we say $t$ is declared in $C$. A type constraint system is defined as follows:

(i) The empty set is a type constraint system.

(ii) If $C$ is a type constraint system and $t \leq \tau$ is a simple type constraint such that $t$ is not declared in $C$ and such that every free variable in $\tau$ is declared in $C$, then $C \cup \{t \leq \tau\}$ is a type constraint system.\(^1\)

Define type constraint derivations of the form $C \vdash \sigma \leq \tau$, for $C$ a type constraint system and $\sigma$, $\tau$ type expressions, from the following set of axioms and rules:

*Type Constraint Axioms.*

\[
C \cup \{t \leq \tau\} \vdash t \leq \tau \\
C \vdash \tau \leq \tau
\]

*Type Constraint Rules.*

\[
C \vdash \rho \leq \sigma, C \vdash \sigma \leq \tau \\
\frac{C \vdash \rho \leq \tau}{C \vdash \sigma \leq \tau}
\]

\[
C \vdash \sigma' \leq \sigma, C \vdash \tau \leq \tau' \\
\frac{C \vdash \sigma' \rightarrow \tau \leq \sigma' \rightarrow \tau'}{C \vdash \sigma \rightarrow \tau \leq \sigma \rightarrow \tau'}
\]

For all $1 \leq j \leq m$, $C \vdash \sigma_j \leq \tau_j$

\[
C \vdash (I_1 : \sigma_1, \ldots, I_m : \sigma_m, \ldots, I_n : \sigma_n) \leq (I_1 : \tau_1, \ldots, I_n : \tau_n)
\]

for $m \leq n$.

The type assignment axioms for this language are only slightly more complex than for the simple typed lambda calculus. In this section we will work in a language where the coercion from a subtype to a type is handled implicitly. In the higher order case discussed later, we will handle this explicitly. A syntactic type assignment, $A$, is a finite set of the form

\[
A = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}
\]

with no variable $x_i$ appearing more than once in $A$. Define type assignment derivations of the form $C, A \vdash e : \sigma$, for $C$ a type constraint system, $A$ a syntactic type assignment, $e$ a term of the typed lambda calculus, and $\sigma$ a type, from the following set of axioms and rules:

\(^1\) Note that we must assume our language has at least one type constant, since otherwise the empty set will be the only type constraint system.
Type Assignment Axioms.

\[ C, A \cup \{ x : \tau \} \vdash x : \tau \]

\[ C, A \vdash c^\tau : \tau \]  \hspace{1cm} (where \( c^\tau \) is a typed constant of type \( \tau \)).

Type Assignment Rules.

\[ C, A \cup \{ x : \sigma \} \vdash e : \tau \]

\[ C, A \vdash \text{fun}(x : \sigma), e : \sigma \rightarrow \tau \]

\[ C, A \vdash e : \sigma \rightarrow \tau, C, A \vdash e' : \sigma \]

\[ C, A \vdash (ee') : \tau \]

\[ C, A \vdash e_i : \tau_i \text{ for } i = 1, \ldots, n \]

\[ C, A \vdash (I_1 = e_1, \ldots, I_n = e_n) : (I_1 : \tau_1, \ldots, I_n : \tau_n) \]

\[ \text{(rec) } \]

\[ C, A \vdash e : (I_1 : \tau_1, \ldots, I_n : \tau_n) \]

\[ \text{ for } i = 1, \ldots, n. \]

The legal terms of the language with respect to a collection of simple type constraints \( C \) and syntactic type assignment \( A \) are those terms \( e \) of the language for which there is a type expression \( t \) such that \( C, A \vdash e : t \). We will only provide a semantics for the legal terms (with respect to some \( C, A \)).

The rules above will form the first order core of the language Minimal Bounded Fun (see Section 4.3). That language will be extended into two relevant languages, for the purposes of our discussion. They will differ in the formal description of a crucial phenomenon: how to consider a term with a given type as a member of a larger type (this is related to the notions of subtype and inheritance in object-oriented languages). Indeed, Bounded Fun formalizes this aspect by extending Minimal Bounded Fun by the following subsumption rule:

\[ \text{(sub) } \]

\[ C, A \vdash e : \sigma, C \vdash \sigma \leq \tau \]

\[ C, A \vdash e : \tau \]

The natural coercion functions discussed above allow one to create models in which the rule (sub) above is sound in a weak sense, i.e., by coercing the meaning of \( e \) from an element of the type \( \sigma \) to an element of type \( \tau \) (see Sections 3 and 5). These will be the models of Coerced Bounded Fun, the other extension of Minimal Bounded Fun, where (sub) is replaced by a formal description of this weak sense, via coercions. It will be important to note that this “natural” coecer need not be exactly the identity function. We will see this in the PER model discussed below, where this coecer is
derived naturally from the identity function, but is not itself the identity (or even injective). However, by this understanding of (sub), it will be also possible to give meaning to (sub) in the stronger (or literal) sense, as described at the end of Section 4.

We note here that Reynolds (1980) seems to be the earliest author to consider the use of non-injective maps to model coercions in computer science. He used these coercions to examine the use of implicit conversions and generic (overloaded) operators from a category-theoretic point of view. He argues that these implicit conversions should behave as homomorphisms with respect to generic operators. Bruce and Wegner (1987) introduce a similar use of coercions to model subtype and inheritance in providing an algebraic model of subtypes and inheritance. This latter paper provided the starting point for the approach in this paper.

3. PER Models

In this section we present the fundamental ideas behind a semantics of the language described in Section 2. The point of this section is to introduce the use of pers as a model of the type structure of a language, with special emphasis on how pers can be used to model subtypes. Since this section is primarily intended to provide an intuitive introduction to the ideas used in more complex settings later in the paper, no attempt is made to carefully specify the formal semantics of terms here. This is done in great detail in Sections 4 and 5.

Let \((0, \cdot)\) be Kleene's applicative structure, i.e., \(n \cdot p\) is the \(n\)th partial recursive function applied to \(p\). Recall that \(A\) is a per on \(\omega\) iff \(A\) is a symmetric and transitive binary relation on \(\omega\). If \(A\) is a per, let \(\text{dom}(A) = \{n \mid nAn\}\). Notice that \(A\) is a (total) equivalence relation on \(\text{dom}(A)\). For \(n \in \text{dom}(A)\), let \(\{n\}_A\) be the equivalence class of \(n\) with respect to \(A\). Let \(Q(A) = \{\{n\}_A \mid n \in \text{dom}(A)\}\), the quotient set of \(\omega\) with respect to \(A\).

3.1. Definition. The category \(\text{PER}\) (of partial equivalence relations on \(\omega\)) has as

**objects:** \(A \in \text{PER}\) iff \(A\) is a symmetric and transitive relation on \(\omega\).

**morphisms:** \(f \in \text{PER}[A, B]\) iff \(f: Q(A) \rightarrow Q(B)\) and

\[
\exists n \forall p (pAp \Rightarrow f(\{n\}_A) = \{n \cdot p\}_B).
\]

Morphisms in \(\text{PER}\) are "computable" in the sense that they are fully described by partial recursive functions which are total on the domain of
the source relation. \( A \) is a discrete per if for all \( n \in \text{dom}(A) \), \( \{n\}_A \) is the singleton set \( \{n\} \).

The types of our model will be partial equivalence relations. If \( A \) is a type (per) then the set of elements of type \( A \) is given by \( Q(A) \). That is, the elements of a type \( A \) are equivalence classes with respect to \( A \).

In order to interpret types correctly we must indicate how to interpret arrow and record types.

**Function spaces.** Let \( A, B \), be pers. Define \( A \rightarrow B \) to be the per such that:

\[
\forall m, n, m(A \rightarrow B) n \Leftrightarrow \forall p, q (pAq \Rightarrow m \cdot pBn \cdot q).
\]

If \( n \in \text{dom}(A \rightarrow B) \), \( \{n\}_{A \rightarrow B} \in Q(A \rightarrow B) \) "represents" a function \( f \) from \( Q(A) \) to \( Q(B) \) such that for all \( \{p\}_A \in Q(A) \), \( f(\{p\}_A) = \{n \cdot p\}_B \). That is, \( n \) represents a function on \( \omega \) which preserves equivalence classes of \( A \).

**Record spaces.** Let \( D = \{d_j \mid j \in J\} \) be a set of natural numbers indexed by elements of a set \( J \). By a slight abuse of notation, we will also use \( D \) to denote the discrete per with domain \( D \). Let \( C_j \), for \( j \in J \), be objects of \( \text{PER} \). Define then

\[
m \left[ \prod_{D} C_j \right] n \Leftrightarrow \forall j \in J \quad (m \cdot d_j) C_j(n \cdot d_j).
\]

As for function spaces, if \( n \in \text{dom}(\prod_{D} C_j) \), \( \{n\}_A \) "represents" a function \( f \) from \( D \) to \( \bigcup_{j \in J} Q(C_j) \) such that for all \( d_j \in D \), \( f(d_j) = \{n \cdot d_j\}_C \). Note that since \( D \), as a per, is discrete, the above definition can be taken as a simple generalization of the definition for function spaces (more on this in Section 5.2).

**3.2. Remark.** \( \text{Dom}[\prod_{D} C_j] = \bigcap_{j \in J} \{n \mid (n \cdot d_j) C_j(n \cdot d_j)\} = \{n \mid \forall j \in J \quad (n \cdot d_j) C_j(n \cdot d_j)\} = \{n \mid \forall j \in J \quad (n \cdot d_j) \in \text{dom}(C_j)\} \).

The arrow types are interpreted over \( \text{PER} \) by function spaces, as usual. For record types, let \( D = \{d_j \mid j \in J\} \) be the interpretation of the labels in \( L' = \{I_j \mid j \in J\} \). Then, given a record type \( A = (I_1 : A_1, \ldots, I_k : A_k) \), such that the interpretation of each \( A_j \) in \( \text{PER} \) is \( C_j \), define the interpretation of \( A \) to be \( [\prod_{D} C_j] \) (see Section 5 to understand this as a dependent or indexed product).

The subtype relation is interpreted as follows.

**3.3. Definition.** Given objects \( B \) and \( C \) in \( \text{PER} \), define \( B \leq C \) by \( B \leq C \) as sets of ordered pairs. I.e., for all \( m, n \), if \( mBn \) then \( mCn \).

Equivalently, \( B \leq C \) iff for all \( n \in \text{dom}(B) \), \( \{n\}_B \subseteq \{n\}_C \). In other words, the partial partition of \( \omega \) given by \( B \) is "finer" than the partial partition
given by \( C \). It is interesting to note that even though subtype is defined in terms of subset of \( \text{pers} \) represented as ordered pairs, the set of elements of the subtype \( B \) is not a subset of the set of elements of the supertype \( C \). In particular, if \( \{n\}_B \in \mathcal{Q}(B) \), then the "natural" coerfer from \( B \) to \( C \) mentioned in Section 2 takes \( \{n\}_B \) to \( \{n\}_C \), where, as noted above, \( \{n\}_B \subseteq \{n\}_C \). Clearly, \( \{n\}_C \) is unique; thus, one may define \( \text{coerce}_{BC} : B \rightarrow C \) by

\[
\text{coerce}_{BC}(\{n\}_B) = \{n\}_C.
\]

Notice that \( \text{coerce}_{BC} \) is actually a morphism in \( \text{PER} \), as it is computed by any index (program) for the identity function. This fact makes it "natural" as a non-injective "embedding."

In the Introduction we provided some intuition as to why this definition of subtype is very natural. We need now to prove that the inference rules in Section 2 are sound with respect to this interpretation. The following theorem takes care of the non-trivial type constraint rules, in particular for function spaces and records (the last of which would fail if records were interpreted as ordinary products).

**3.4. Theorem.** (i) \( C' \leq C \) and \( E \leq E' \) imply \( C \rightarrow E \leq C' \rightarrow E' \).

(ii) If \( D' = \{d_i | i \in I\} \subseteq D = \{d_j | j \in J\} \), such that \( \forall i \in I, C_i \leq C_i' \), then

\[
\left[ \prod_{D} C_i \right] \leq \left[ \prod_{D'} C_i' \right].
\]

**Proof.** (i) \( \forall m, n, m(C \rightarrow E)n \Leftrightarrow \forall p, q \ (pCq \Rightarrow (m \cdot p) E(n \cdot q) \Rightarrow \forall p, q \ (pC'q \Rightarrow (m \cdot p) E'(n \cdot q) \Rightarrow m(C' \rightarrow E')n. \)

(ii) \( m[\prod_{D} C_i]n \Leftrightarrow \forall j \in J \ (m \cdot d_j) C_j(n \cdot d_j) \Rightarrow \forall i \in I \ (m \cdot d_i) C'_i(n \cdot d_i) \Rightarrow m[\prod_{D'} C'_i]n. \)

The result also follows from Theorem 5.2.3 in Section 5.

One may also give a formal definition of the meaning of terms and show that the type assignment axioms and rules are sound. As for the rules, note that (sub) is not quite sound as it is. One has to use coercions, i.e., \( b : B \leq C \) implies, for \( b = \{n\}_B \), \( \text{coerce}_{BC}(\{n\}_B) = \{n\}_C \) which is an element of type \( C \).

As for the other rules, they are interpreted as in the usual quotient-set semantics of typed \( \lambda \)-calculus. Details will be given in Sections 4 and 6. Briefly, \( e \cdot e' : \tau \) is described as the application of two equivalence classes, \( \{n\}_{B \rightarrow C} \) and \( \{m\}_B \), say, by setting

\[
\{n\}_{B \rightarrow C} \cdot \{m\}_B = \{n \cdot m\}_C.
\]
Fun \((x : \sigma). e : \sigma \rightarrow \tau\) is interpreted as a morphism between the two types, or, more precisely, as the equivalence class of indices of its computations.

Now that we have completed an informal survey of the per-based semantics of this first-order typed language, in the next section we begin a more formal and complete examination of the syntax and semantics of the more complex language Bounded Fun.

4. Syntax and Semantics of Bounded Fun

In this section we develop the formal syntax and semantics of the language Bounded Fun described in CW (1985). Bounded Fun is an extension of the second-order lambda calculus (itself invented independently by Girard, 1972, and Reynolds, 1974), which supports subtypes and inheritance. Readers are referred to CW (1985) for examples of the expressibility of that language. The language defined here is modified slightly from the presentation in CW (1985). The definition of the models of this language is based on that for the second-order lambda calculus given in Bruce et al. (1990). The presentation, thus, has an elementary set-theoretic flavor, which avoids the difficulties (and depth) of the categorical approaches at the price of some technicalities. We believe that this may be more appealing to most computer scientists. For informative categorical approaches, based on relevant categorical structures, one should consult the work of Seely (1986) and Moggi (see Asperti and Longo, 1989).

In Sections 4.1 and 4.2 we develop the syntax and semantics of the language without the rule (sub) introduced in Section 2. We will call this language Minimal Bounded Fun. In Section 4.3 we introduce an extension of the language with a constant \texttt{convert} and axioms and rules which govern its behavior. We call this language Coerced Bounded Fun. Finally, we introduce the semantics of the usual Bounded Fun (i.e., Minimal Bounded Fun with the added rule (sub)) by relating it to that of Coerced Bounded Fun.

4.1. The Syntax of Minimal Bounded Fun

We begin with a description of the language. As noted above, the only difference between the language introduced in this section and usual Bounded Fun is the omission of the type inference rule (sub). In Section 4.3, we return to the original system.

We note that the definitions of kind expressions and constructor expressions are mutually recursive, since type expressions (a subset of the constructor expressions) appear in kind expressions. We use the notation \(e[a/b]\) to denote the expression formed by replacing all free occurrences of \(b\) in \(e\) by \(a\) (where the names of bound variables of \(e\) are changed while...
necessary to avoid capturing free variables of \( a \). The formal definition is left as an exercise to the reader.

**Kind expressions.** In the following the reader should think of \( T \) as representing the collection of all types, and for each type \( \tau \), \( T \leq \tau \) as the set of all types less than or equal to \( \tau \). Let \( L \) be a countable collection of labels, \( \{ I_1, I_2, \ldots \} \), and for each \( s \subseteq \omega \), let \( L_s = \{ I_j \mid j \in s \} \). Other kind expressions will denote other classes of higher order objects such as functions from one kind to another and functions from types to kinds.

The set of all **kind expressions**, \( \kappa \), is defined as follows:

\[
\kappa ::= T | T \leq \tau | L_s | \kappa_1 \rightarrow \kappa_2 | \prod_{t:T} \kappa,
\]

where \( T \) and \( T \leq \) are special symbols, \( \tau \) is a type expression (the definition of type expressions is given below), \( t \) is a constructor variable of kind \( T \) (i.e., a type variable—see below), and \( s \subseteq \omega \).

**Constructor expressions.** Constructor expressions are used to build objects of various kinds. The most commonly used constructor expressions are those which are used to construct new types from other types or functions on types. Examples are given after the definition.

Let \( \mathcal{C}_{\text{con}} \) be a collection of constructor constant symbols, with associated kinds, and \( \mathcal{V}_{\text{con}} \) be an infinite collection of constructor variable symbols, with associated kinds. Read \( \mu : \kappa \) as \( \mu \) is a constructor expression of kind \( \kappa \). The **constructor expressions** (with their associated kinds) are defined as follows:

(i) \( c^\kappa : \kappa \) for \( c^\kappa \in \mathcal{C}_{\text{con}} \), \( v^\kappa : \kappa \) for \( v^\kappa \in \mathcal{V}_{\text{con}} \).

(ii) If \( \mu : \kappa \Rightarrow \kappa' \), \( v : \kappa \) then \( (\mu v) : \kappa' \).

(iii) If \( \mu : \prod_{t:T} \kappa \), \( \rho : T \) then \( (\mu \rho) : \kappa[\rho/t] \).

(iv) If \( \mu : \kappa' \) then \( \lambda v^\kappa : \mu : \kappa \Rightarrow \kappa' \), if \( v^\kappa \) does not occur free in \( \kappa' \).

(v) If \( \mu : \kappa' \) then \( \pi t. \mu : \prod_{t:T} \kappa \), if \( t \in \mathcal{V}_{\text{con}}, t : T \).

(vi) If \( j \in s \subseteq \omega \) then \( I_j : L_s \).

We will say that \( \tau : T \) is a **type expression**. We will assume that there are an infinite number of constant constructor symbols in our language. These must include:

(i) \( \to : T \Rightarrow (T \Rightarrow T) \) (written in infix style).

(ii) \( \forall : (T \Rightarrow T) \Rightarrow T \), and \( \forall \leq : \prod_{t:T} ( (T \leq \tau \Rightarrow T) \Rightarrow T ) \).

(Note that we will usually write \( \forall t. \sigma \), rather than \( \forall (\lambda t^T. \sigma) \), and \( \forall t \leq \tau. \sigma \), rather than \( (\forall \leq \tau)(\lambda t^T \cdot \tau. \sigma) \).)

(iii) For each finite \( s \subseteq \omega \), a constructor, \( R_s : (L_s \Rightarrow T) \Rightarrow T \).
If \( L_s = \{J_1, ..., J_n\} \) and \( F : L_s \Rightarrow T \), where for each \( J_i \), \( F(J_i) = \tau_i \), then \( R_xF \) will usually be written \( (J_1 : \tau_1, ..., J_n : \tau_n) \), as in Section 2.

Although we have not specified it here, we presume that there is a mechanism in the language to specify all functions in \( L_s \Rightarrow T \) for each finite \( s \subseteq \omega \). For example, if \( L_s = \{J_1, ..., J_n\} \) there might be a constructor constant \( \tau_1 \cdots \tau_n \) such that \( F \) denotes the function which takes each \( J_k \) to \( \tau_k \).

If \( C \) is a collection of simple type constraints, then we can derive other type constraints using the axioms and rules in Section 2 plus the following rules:

**Type Constraint Rules.**

\[
\begin{align*}
C &\vdash \sigma \leq \sigma', C \cup \{t \leq \sigma'\} &\vdash \tau \leq \tau' \text{ for } t \text{ not free in } C \\
C &\vdash \forall t. \sigma, \tau \leq \forall t. \sigma', \tau' \\
C &\vdash \tau \leq \tau' \text{ for } t \text{ not free in } C \\
C &\vdash \forall t. \tau \leq \forall t. \tau'.
\end{align*}
\]

Note that these are relatively straightforward generalizations of the rules for function spaces given in Section 2.

**Terms.** We begin our description of the terms of the language by first defining pre-terms. These are expressions which may not be typeable, and hence not all of these will be meaningful in our models. Let \( \mathcal{Y}_{\text{term}} \) be an infinite collection of variables and \( \mathcal{C}_{\text{term}} \) be a set of constants, each of the constants having a fixed closed type.

4.1.1. **Definition.** The set \( \text{PreTerm} \) of pre-terms is defined by:

\[
e ::= c \mid x \mid \text{fun}(x : \tau). e \mid (ee') \mid \text{all } t^\tau. e \mid \text{all } t^{\tau} < \sigma^\tau. e \mid e[\sigma]\mid (J_1 = e_1, ..., J_n = e_n) \mid e. I,
\]

where \( c \in \mathcal{C}_{\text{term}}, x \in \mathcal{Y}_{\text{term}}, \sigma, \tau \) are type expressions, and \( J_1, ..., J_n, I \) are identifiers from \( L \).

For simplicity we will abbreviate terms of the form \( \text{all } t^\tau. e \) by dropping the superscript on the type variable: \( t. e \), and we abbreviate terms of the form \( \text{all } t^{\tau} < \sigma^\tau. e \) by dropping the superscript and writing the type constraint more explicitly: \( (t \leq \tau). e \).

In order to determine whether pre-terms are type-correct, we must first assign types to the free variables occurring in terms. Let \( A \) be a syntactic type assignment. The type assignment axioms and rules for Minimal Bounded Fun are as in Section 2 plus the following:
Type Assignment Rules.

\[ C \cup \{ t \leq \sigma \}, A \vdash e : \tau \text{ for } t \text{ not free in } A \text{ or } C \]
\[ C, A \vdash \text{all}(t \leq \sigma). e : \forall t \leq \sigma. \tau \]

\[ C, A \vdash e : \tau \text{ for } t \text{ not free in } A \text{ or } C \]
\[ C, A \vdash \text{all } t. e : \forall t. \tau \]

\[ C, A \vdash e : \forall t \leq \sigma'. \tau, C \vdash \sigma \leq \sigma' \]
\[ C, A \vdash e[\sigma] : \tau[\sigma/t] \]

\[ C, A \vdash e[\sigma] : \tau[\sigma/t] \]

We write \( C, A \vdash_m e : \sigma \) for type inference in Minimal Bounded Fun. We say that a pre-term \( e \) is a term of Minimal Bounded Fun with respect to \( C, A \) if there is a type expression \( \sigma \) such that \( C, A \vdash_m e : \sigma \). We indicate the existence of such a \( \sigma \) by writing \( e \in \text{MBF}_{C,A} \).

The conversion axioms and rules which correspond to the operational semantics of the language are variants of the usual (\( \alpha \)) and (\( \eta \)) axioms with associated rules for the typed lambda calculus. Readers who are familiar with the second-order lambda calculus, will have no trouble writing down the appropriate variants here. (Note that there is one version of each of (\( \alpha \)) and (\( \beta \)) for each of the three variable binding operations). It is also easy to formulate appropriate versions of the (\( \eta \)) axiom for this language (this axiom guarantees that each of the kinds of functions is extensional). The model given Section 5 will also satisfy all of the (\( \eta \)) axioms.

4.2. The Semantics of Minimal Bounded Fun

Our definition of models for Minimal Bounded Fun is based on that of the environment models for the second-order lambda calculus given in Bruce et al. (1990). The main conceptual addition necessary for Minimal Bounded Fun is a partial ordering on types which satisfies the types constraint rules. Elements of record types are interpreted as functions from the field labels to elements of the corresponding types (as in Section 2), while (bounded) polymorphic functions are interpreted as functions from (bounded sets of) types to elements of the corresponding types.

Since the material in the next two sections is rather technical, the reader is strongly urged simply to skim these sections of the paper on a first reading. Section 5 should be mainly understandable with only a cursory knowledge of this section on the definition of formal models.

The semantic structures for our higher order objects, the constructor expressions, are based on the definition of models for the simple typed lambda calculus in which the kind expressions replace type expressions and the constructor expressions replace typed terms.
A **kind frame** $\text{Kind}$ for a set $C_{\text{est}}$ of constructor constants is a tuple

$$
\text{Kind} = \left\langle \text{Kinds}, \{\text{Kind}^k | k \in \text{Kinds}\}, \{\Phi_{k, k'} | k, k' \in \text{Kinds}\}, \langle \Phi_{f} | f \in \text{Type} \to \text{Kinds} \rangle, I_{\text{Kind}} \right>, \prod_{T}, \leq, \right>,
$$

where $\langle \{\text{Kind}^k | k \in \text{Kinds}\}, \{\Phi_{k, k'} | k, k' \in \text{Kinds}\}, I_{\text{Kind}} \rangle$ is essentially a model of the typed lambda calculus, with

1. $\text{Kinds}$, a set closed under $\Rightarrow$, and $\prod_{T} f$ for $f \in \text{Type} \to \text{Kinds}$,

2. Each $\text{Kind}^k$ represents the set of all constructors with kind $k$.

3. Each $\Phi_{k, k'} : \text{Kind}^k \Rightarrow k' \to (\text{Kind}^k \to \text{Kind}^{k'})$ is an injection which allows each element of $\text{Kind}^k \to k'$ to be interpreted as a function,

4. For $f \in \text{Type} \to \text{Kinds}$, $\Phi_{f} : \text{Kind}^{\prod_{T} f} \to (\prod_{a \in \text{Type}} f(a))$ is an injection which allows each element of $\text{Kind}^{\prod_{T} f}$ to be interpreted as a function,

5. $I_{\text{Kind}}$ is a function giving the denotation of constructor and kind constants. Let $T$ abbreviate $I_{\text{Kind}}(T)$ and Type abbreviate $\text{Kind}^{T}$. Let $T_{\equiv} : \text{Type} \to \text{Kinds}$ be defined so that for $b \in \text{Type}$, $k = T_{\equiv} b$ implies that $\text{Kind}^k = \{a \in \text{Type} | a \leq b\}$ (such a $k$ must exist for each $b$ by (7) below). For $a \in \text{Type}$, let Type $\leq a$ abbreviate $\text{Kind}^{\leq a}$. Let $L_{s} = I_{\text{Kind}}(L_{s})$.

6. $\leq$ is a partial ordering on Type which satisfies:
   
   i. If $a' \leq a$ and $b \leq b'$ then $a \to b \leq a' \to b'$.
   
   ii. If $F : \text{Kind}^{L_{r} \Rightarrow T}, \ G : \text{Kind}^{L_{r} \Rightarrow T}$, where $r \subseteq s$, and for all $j \in r$,

   $\Phi_{L_{r}, T}(F)(j) \leq \Phi_{L_{r}, T}(G)(j)$, then $\text{R}, F \leq \text{R}, G$.

   iii. If $F \in \text{Kinds}^{(T_{\equiv} b) \Rightarrow T}, \ G \in \text{Kinds}^{(T_{\equiv} b') \Rightarrow T}$ such that $b' \leq b$ and for all $a \leq b'$,

   $\Phi_{T_{\equiv} b, T}(F)(a) \leq \Phi_{T_{\equiv} b', T}(G)(a)$, then $\forall_{\leq}(b)(F) \leq \forall_{\leq}(b')(G)$.

   iv. If $F \in \text{Kind}^{T \Rightarrow T}, \ G \in \text{Kind}^{T \Rightarrow T}$ such that for all $a \in \text{Type}$,

   $\Phi_{T, T}(F)(a) \leq \Phi_{T, T}(G)(a)$, then $\forall F \leq \forall G$.

7. For each $b \in \text{Type}$ there is a $k \in \text{Kinds}$ such that $\text{Kind}^k = \{a \in \text{Type} | a \leq b\}$.
Interpretations of constructor expressions are as for the simply typed lambda calculus. Let $\eta$ be an environment which assigns constructor variables to elements of corresponding kinds in $\bigcup_{k \in \text{Kind}} \text{Kind}^k$. $\eta[a/v]$ is defined as that environment which is identical to $\eta$ except that it takes the value $a$ on $v$. We define the meaning of kind expressions as

$$\llbracket T \rrbracket \eta = T,$$

$$\llbracket T \text{ or } \tau \rrbracket \eta = T \text{ or } (\llbracket \tau \rrbracket \eta),$$

$$\llbracket L_s \rrbracket \eta = L_s,$$

$$\llbracket \kappa_1 \Rightarrow \kappa_2 \rrbracket \eta = \llbracket \kappa_1 \rrbracket \eta \Rightarrow \llbracket \kappa_2 \rrbracket \eta,$$

$$\llbracket \prod_{I \in T} \kappa \rrbracket \eta = \prod_{I \in T} (\lambda a \in T. \text{Kind}^k \eta[a/I]),$$

and the meaning of constructor expressions as

$$\llbracket v^\kappa \rrbracket \eta = \eta(v^\kappa),$$

$$\llbracket c^\kappa \rrbracket \eta = I_{\text{Kind}}(c^\kappa), \llbracket I_j \rrbracket \eta = I_{\text{Kind}}(I_j)$$

$$\llbracket \mu \nu \rrbracket \eta = \Phi_{k,k'}(\llbracket \mu \rrbracket \eta) \llbracket \nu \rrbracket \eta \quad \text{for} \quad \mu : \kappa \Rightarrow \kappa', k = \llbracket \kappa \rrbracket \eta, k' = \llbracket \kappa' \rrbracket \eta,$$

$$\llbracket \mu \rho \rrbracket \eta = \Phi_f(\llbracket \mu \rrbracket \eta) \llbracket \rho \rrbracket \eta,$$

where $f(a) = \llbracket \kappa \rrbracket \eta[a/I], \quad a \in \text{Type}, \mu : \prod_{I \in T} \kappa,$

$$\llbracket \lambda v^\kappa. \mu \rrbracket \eta = \Phi_{k,k'}(f), \quad \text{where} \quad f(a) = \llbracket \mu \rrbracket \eta[a/v^\kappa]$$

for all $a \in \text{Kind}^k$, $\mu : \kappa', k = \llbracket \kappa \rrbracket \eta, k' = \llbracket \kappa' \rrbracket \eta,$

$$\llbracket \pi t. \mu \rrbracket \eta = \Phi_{f^{-1}}(g), \quad \text{where} \quad g(a) = \llbracket \mu \rrbracket \eta[a/t] \quad \text{for all} \quad a \in \text{Type}, \mu : \kappa,$$

and for $a \in \text{Type}, f(a) = \llbracket \kappa \rrbracket \eta[a/I]$.

Notice that $I_{\text{Kind}}$ must give meanings to $\rightarrow, \forall, \forall^\omega$, and $R$, for each $s \subseteq \omega$. For notational simplicity we use the same symbol for these syntactic objects and their meanings. We say that an environment $\eta$ is well-kinded with respect to $\text{Kind}$ if for each constructor variable, $v^\kappa, \eta(v^\kappa) \in \text{Kind}^k \eta$. A kind model for $\mathcal{E}_{\text{est}}$ is a kind frame for $\mathcal{E}_{\text{est}}$ in which every constructor expression has a meaning with respect to all well-kinded environments.

Environment models for Minimal Bounded Fun will be defined using kind structures to interpret kinds and constructors, with added domains to interpret the terms.
4.2.2. Definition. A frame $F$ for Minimal Bounded Fun with constants from $\mathcal{C}_\text{est}$ and $\mathcal{C}_\text{term}$ is a tuple

$$F = \langle \text{Kind}, \text{Dom}, \{\Phi_{a,b}\}, \{\Phi_F\}, \{\Phi_{F,b}\} \rangle$$

which satisfies conditions (i) through (vi) below:

(i) $\text{Kind} = \langle \text{Kinds}, \{\text{Kind}^k | k \in \text{Kinds}\}, \{\Phi_{k,k'} | k, k' \in \text{Kinds}\}, \{\Phi_f | f \in \text{Type} \to \text{Kinds}\}, I_{\text{ Kind}}, \sim, \prod \rangle$ is a kind model for $\mathcal{C}_\text{est}$.

(ii) $\text{Dom} = \langle \{\text{Dom}^a | a \in \text{Type}\}, I_{\text{Dom}} \rangle$, where $I_{\text{Dom}} : \mathcal{C}_\text{term} \to \bigcup_{a \in \text{Type}} \text{Dom}^a$ such that for each $c_i^\tau \in \mathcal{C}_\text{term}$, $I_{\text{Dom}}(c_i^\tau) \in \text{Dom}^{[\tau]}$.

(iii) For each $a, b \in \text{Type}$, there is a set $[\text{Dom}^a \to \text{Dom}^b]$ of functions from $\text{Dom}^a$ to $\text{Dom}^b$ with $\Phi_{a,b} : \text{Dom}^a \to \text{Dom}^b$ a bijection.

(iv) For each $F \in \text{Kind}^{T \Rightarrow T}$ and $F = \Phi_{T,T}(F)$, there is a subset $[\prod a \in \text{Type} \text{ Dom}^{F(a)}] \subseteq [\prod a \in \text{Type} \text{ Dom}^{a}]$ with $\Phi_F : \text{Dom}^{\forall a \in \text{Type} \text{ Dom}^{F(a)}}$ a bijection. Here $\forall = \Phi_{T \Rightarrow T}(\forall)$.

(v) For each $b \in \text{Type}$ and $F \in \text{Kind}^{T \Rightarrow b}$ and $F = \Phi_{T,b,T}(F)$, there is a subset $[\prod a \in \text{Type} \text{ Dom}^{F(a)}] \subseteq [\prod a \in \text{Type} \text{ Dom}^{a}]$ with $\Phi_{F,b} : \text{Dom}^{\forall a \in \text{Type} \text{ Dom}^{F(a)}}$ a bijection. Here $\forall = \Phi_{T \Rightarrow T}(\forall)$.

(vi) For each $F \in \text{Kind}^{L \Rightarrow T}$ and $F = \Phi_{L,T}(F)$, there is a subset $[\prod i \in Ls \text{ Dom}^{F(i)}] \subseteq [\prod i \in Ls \text{ Dom}^{F(i)}]$ with $\Phi_{F} : \text{Dom}^{R,F} \to [\prod i \in L \text{ Dom}^{F(i)}]$ a bijection. Here $R_i = \Phi_{L,T}(\forall, \forall)$.

Condition (iii) states that $\text{Dom}^a \to \text{Dom}^b$ must “represent” some set $[\text{Dom}^a \to \text{Dom}^b]$ of functions from $\text{Dom}^a$ to $\text{Dom}^b$. Similarly, conditions (iv), (v), and (vi) specify that the other complex domains must represent some appropriate collections of functions. Note in particular that by our abbreviations, (vi) implies that $(\Phi_{1}, \ldots, \Phi_{n})$ represents a collection of functions, $g$, such that for $1 \leq i \leq n$, $g(I_{\text{Kind}}(J_i)) \in \text{Dom}^{[\tau_i]}$. (We offer a note of reassurance to the reader who has made it this far. The rest of the section is rather straightforward.)

Frames have exactly the right structure to define the meaning of terms of Minimal Bounded Fun. Let $A$ be a syntactic type assignment. Let $\eta$ be an environment mapping $\forall^\tau$ to elements of the appropriate kinds, and $\forall^\tau$ to elements of $\bigcup_{a \in \text{Type}} \text{Dom}^a$. We say that $\eta$ satisfies $C, A$, written $\eta \models C, A$, if $\eta(x) \in [A(x)]^\eta$ for each variable $x \in \text{dom}(A)$, and $\eta(t) \subseteq [\tau] \eta$ for each $t \in \tau$ in $C$.

4.2.3. Definition. Let $F$ be a second-order functional domain for Minimal Bounded Fun, let $A$ be a syntactic type assignment, and $C$ a collection of simple type constraints. If $\eta \models C, A$ then the meanings of terms of Minimal Bounded Fun can be defined inductively on the derivation of typings as follows:
\[
\llbracket C, A \vdash_m x : \tau \rrbracket \eta = \eta(x),
\]
\[
\llbracket C, A \vdash_m c' : \tau \rrbracket \eta = I_{dom}(c'),
\]
\[
\llbracket C, A \vdash_m (ee') : \tau \rrbracket \eta = (\Phi_{a,b}\llbracket C, A \vdash_m e : \sigma \rightarrow \tau \rrbracket \eta)\llbracket C, A \vdash_m e' : \sigma \rrbracket \eta,
\]
where \( [\sigma] \eta = a \) and \( [\tau] \eta = b \),
\[
\llbracket C, A \vdash_m \text{fun}(x : \sigma), e : \sigma \rightarrow \tau \rrbracket \eta = \Phi_{a,b}^{-1} g,
\]
where \( g(d) = \llbracket C, A \cup \{ x : \sigma \} \vdash_m e : \tau \rrbracket \eta[d/x] \) for all \( d \in \text{Dom}^\sigma \),
\[
\llbracket C, A \vdash_m e. J_i : \tau_i \rrbracket \eta = \Phi_F(\llbracket C, A \vdash_m e : (J_1 : \tau_1, \ldots, J_n : \tau_n) \rrbracket \eta)(\llbracket J_i \rrbracket \eta),
\]
where \( F \in \text{Kind}_L \rightarrow T \) such that for \( 1 \leq i \leq n, \Phi_{L,a}(F)(\llbracket J_i \rrbracket \eta) = \llbracket \tau_i \rrbracket \eta \),
\[
\llbracket C, A \vdash_m (J_1 = e_1, \ldots, J_n = e_n) : (J_1 : \tau_1, \ldots, J_n : \tau_n) \rrbracket \eta = \Phi_{F^{-1}} g
\]
where \( F \in \text{Kind}_L \rightarrow T \) such that for \( 1 \leq i \leq n, \Phi_{L,a}(F)(\llbracket J_i \rrbracket \eta) = \llbracket \tau_i \rrbracket \eta \)
and \( g(\llbracket J_i \rrbracket \eta) = \llbracket C, A \vdash_m e_i : \tau_i \rrbracket \eta \),
\[
\llbracket C, A \vdash_m e[\sigma] : \tau[\sigma/t] \rrbracket \eta = (\Phi_{F,b}\llbracket C, A \vdash_m e : \forall t \leq \sigma \rightarrow \tau \rrbracket \eta)(\llbracket \sigma \rrbracket \eta),
\]
where \( F = \llbracket \lambda t \leq \sigma \rightarrow \tau \rrbracket \eta \) and \( b = \llbracket \sigma \rrbracket \eta \),
\[
\llbracket C, A \vdash_m \forall t \leq \sigma, e. \forall t \leq \sigma, \tau \rrbracket \eta = \Phi_{F,b} g,
\]
where \( g(a) = \llbracket C \cup \{ t \leq \sigma \} \vdash_m e : \tau \rrbracket \eta[d/t] \) for all \( a \in \text{Type}^{\leq b} \) and \( F = \llbracket \lambda t \leq \sigma \rrbracket \eta \), where \( b = \llbracket \sigma \rrbracket \eta \).
\[
\llbracket C, A \vdash_m \text{all} t. e : \forall t \leq \sigma, \tau \rrbracket \eta = \Phi_{F^{-1}} g,
\]
where \( g(a) = \llbracket C \vdash_m e : \tau \rrbracket \eta[d/t] \) for all \( a \in \text{Type} \) and \( F = \llbracket \lambda t \rrbracket \eta \).

As the reader no doubt suspects, the semantics given in Definition 4.2.3 is independent of the particular derivation of the typing. This explains why we did not annotate terms with derivations. The following lemma, which corresponds to Lemmas 2 and 8 of Bruce et al. (1990), shows that the above definition is well defined. If \( \mu, \nu \) are constructor expressions, we write \( \vdash_{\text{con}} \mu = \nu \), if the equality is provable in a proof system in which versions of the usual \( (\alpha), (\beta), \) and \( (\eta) \) axioms and corresponding congruence rules are given for the constructor expressions.
4.2.4. **Lemma.** Suppose $\Delta_1, \Delta_2$ are derivations of typings $C, A \vdash_m e : \sigma$ and $C, B \vdash_m e : \tau$, respectively, and $\eta$ is an environment such that:

1. For every $x$ free in $e$, $\longmapsto_{\text{con}} A(x) = B(x)$, and
2. $\eta \models C, A$ and $\eta \models C, B$.

Then

3. $\longmapsto_{\text{con}} \sigma = \tau$, and
4. $\llbracket C, A \vdash_m e : \sigma \rrbracket \eta = \llbracket C, A \vdash_m e : \tau \rrbracket \eta$ (where the first term is evaluated with respect to the typing derivation $\Delta_1$, and the second with respect to $\Delta_2$).

When we begin with $\sigma = \tau$, we get the desired result. The proof is virtually identical to those given for Lemmas 2 and 8 in Bruce *et al.* (1988). The reader is referred to that paper for more details.

4.2.5. **Definition.** A frame $F$ for Minimal Bounded Fun is a model of Minimal Bounded Fun if, for all $\eta \models C, A$ and all $e$ such that $C, A \vdash_m e : O$, $\llbracket C, A \vdash_m e : O \rrbracket \eta$ is defined.

It is easy to verify that every model, $F$, of Bounded Fun satisfies the type constraint and type assignment rules. In particular, if $\eta \models C, A$ and $C, A \vdash_m e : \tau$, then $\llbracket C, A \vdash_m e : \tau \rrbracket \eta \in \text{Dom}^{[\tau]} \eta$. The proofs are only minor variants of the similar rules for the second-order lambda calculus in Bruce *et al.* (1988). Similarly, it is easy to verify the conversion rules (e.g., the variants of $(\alpha)$ and $(\beta)$).

4.3. The Semantics of the Original Bounded Fun

In this section we will provide the semantics of the original Cardelli and Wegner system with the type inference rule (sub). We will see that this introduces much greater complexity to the semantics. This complexity arises since there may be several quite distinct type derivations for the same expression. In particular there may be several distinct typing derivations which result in the assignment of the same type expression to a term. Lemma 4.2.4 becomes much more difficult to prove in the presence of rule (sub).

We choose to approach this problem indirectly by first introducing the polymorphic constant $\text{convert}$, which is used to coerce elements from subtypes to supertypes. After introducing appropriate axioms and rules to govern the behavior of this constant, we show that any typing derivation of a term in the Cardelli and Wegner system corresponds to a typing derivation of a coerced term in our system (which does not include the type inference rule (sub)). Moreover, the "erasure" of this coerced term yields the original term. Finally we show that the meaning of all coerced terms
with the same “erasure” cohere in a way to be made precise later, allowing us to define the meaning of terms in the original system.

Since in Sections 4.1 and 4.2 we have omitted rule (sub) which states that the set of types of a term is closed under supertypes, it is useful to add the new polymorphic constant:

$$\text{convert} : \forall \tau. \forall s \leq t. s \rightarrow t.$$  

The function of convert is to transform a value of type $s$ to type $t$ for $s \leq t$. This constant will allow us to define more flexible record selectors as follows. If $I$ is an identifier then we can define

$$\text{select}_I : \forall \tau. \forall s \leq (I : t). s \rightarrow t$$

by

$$\text{select}_I = \text{all } t. \text{all } (s \leq (I : t)). \text{fun } (x : s). \text{convert } [I : t][s] x.$$  

Thus one can select components of elements which are subtypes of record types.

Note that if $C, A \vdash e : \sigma$, and $C \vdash \sigma \leq \tau$, then $C, A \vdash \text{convert } [\tau][\sigma] e : \tau$. However, we note here that it does not necessarily follow that for an environment $\eta$,

$$\llbracket C, A \vdash e : \sigma \rrbracket \eta = \llbracket C, A \vdash \text{convert } [\tau][\sigma] e : \tau \rrbracket \eta.$$  

In particular, it does not follow that if $C \vdash \sigma \leq \tau$, then $[[\sigma]] \eta \subseteq [[\tau]] \eta$. Thus convert will typically have a real semantic effect.

We will now extend Minimal Bounded Fun to include the constant convert as well as axioms and rules to ensure that it behaves properly.

4.3.1. DEFINITION. Define Coerced Bounded Fun to be the extension of Minimal Bounded Fun obtained by adding the constant, convert, with type $\forall \tau. \forall s \leq t. s \rightarrow t$, to the language and adding the following axioms and rules:

(E1) \hspace{1cm} \text{convert } [\sigma][\sigma] = \text{fun } (x : \sigma). x

(E2) \hspace{1cm} C \vdash \sigma \leq \tau, C \vdash \tau \leq \rho \\
\hspace{3cm} \frac{C, A \vdash (\text{convert } [\rho][\tau]). (\text{convert } [\tau][\sigma]) = \text{convert } [\rho][\sigma]}{}

(E3) \hspace{1cm} C \vdash \sigma' \leq \sigma, \tau \leq \tau', C, A \vdash e : \sigma \rightarrow \tau \\
\hspace{3cm} \frac{C, A \vdash \text{convert } [\sigma' \rightarrow \tau'][\sigma \rightarrow \tau] e \begin{array}{l} = \text{fun } (x : \sigma'). \text{convert } [\tau'][\sigma] (e(\text{convert } [\sigma][\sigma'] x)) \end{array} }{}

(E4) \hspace{1cm} C \vdash \sigma' \leq \sigma, C \cup \{ t \leq \sigma' \} \vdash \tau \leq \tau', C, A \vdash e : \forall \tau. t \leq \sigma, \tau \\
\hspace{3cm} \frac{C, A \vdash \text{convert } [\forall t \leq \sigma'. \tau'][\forall t \leq \sigma. \tau] e \begin{array}{l} = \text{all } (t \leq \sigma'). \text{convert } [\tau'][\tau] (e[t]) \end{array} }{\text{for all } t \leq \sigma'.}
The above axioms and rules were chosen to ensure that convert behaves essentially as a homomorphism (i.e., making certain diagrams commute). We do not know if these are minimal axioms for this system and would hope that (E7) and (E8) in particular could be simplified. These axioms and rules will be used in the proof of Theorem 4.3.7.

As Coerced Bounded Fun extends Minimal Bounded Fun by the constant convert specified by the above axioms and rules, a model for Coerced Bounded Fun is a model for Minimal Bounded Fun which has an interpretation for convert which satisfies its properties. The meanings of terms is as given for Minimal Bounded Fun. Note that since we have only added conversion rules rather than type inference rules, Lemma 4.2.4 still holds.

We next extend Minimal Bounded Fun in another direction: Bounded Fun is obtained from Minimal Bounded Fun by adding the subsumption rule (sub) of Section 2.
4.3.2. **DEFINITION.** Let Bounded Fun be the system whose language, type inference, and proof system is the same as Minimal Bounded Fun, but whose type inference rules also includes the subsumption rule:

\[
\frac{C \vdash \sigma \subseteq \tau, C, A \vdash e : \sigma}{C, A \vdash e : \tau} \tag{sub}
\]

We write \( C, A \vdash_{bf} e = e' \), and \( C, A \vdash_{bf} e : \sigma \) for proofs of equality and type inference in this language to distinguish it from the minimal and coerced systems. We write \( e \in BF_{C,A} \) if there is a \( \sigma \) such that \( C, A \vdash_{bf} e : \sigma \).

An added difficulty in interpreting this system is that terms typically have meanings of several types. Moreover, there may be several proofs that a term has a particular type. For example, if \( C \vdash \sigma \equiv \tau, C, A \vdash e : \tau \rightarrow \rho \), and \( C, A \vdash e : \sigma \rightarrow \rho \) then one can show that \( C, A \vdash_{bf} e : \rho \) by first inferring \( C \vdash \tau \rightarrow \rho \equiv \sigma \rightarrow \rho \), and then \( C, A \vdash e : \sigma \rightarrow \rho \) by the subsumption rule, and finally using the rule for typing function application. Alternatively, one could infer \( C, A \vdash e' : \tau \) by subsumption and then using the rule for typing function applications. Since the meaning of terms is defined by induction on the proof of typing, it is not at all clear that the meaning obtained through these different proofs are the same.

The original version of this paper, Bruce and Longo (1988), did not explicitly address this question. In Breazu-Tannen et al. (1989), this is taken care of by translating terms of Bounded Fun into terms of the second-order lambda calculus and then showing that all possible translations were provably equal. We subsequently decided to approach the problem in a somewhat different fashion, replacing conditions on models that appeared in the previous version of this paper by the explicit axioms on `convert` given in Definition 4.3.1 above. (We note that because of the uniformity of the coercion functions used in the model in Section 5, it is obvious that all possible meanings of a term in the same type are equal.) In what follows, we show that this approach guarantees that the interpretation of a term in a particular type is independent of the proof that it has that type.

We will interpret terms of Bounded Fun as being abbreviations of certain terms of Coerced Bounded Fun. We say a term of Coerced Bounded Fun is **translatable** if all occurrences of the constant `convert` appear in subterms of the form `convert [\sigma][\tau] e`, where \( \sigma \) and \( \tau \) are type expressions and \( e \) is a translatable term of Coerced Bounded Fun. (This is equivalent to treating `convert` as a new term-building operator which takes three arguments, two types and the third a term. We choose this alternative approach so that we can use the model definition given in Section 4.2 without having to redo all the work in that section.)

4.3.3. **DEFINITION.** Let \( e \) be a translatable term of Coerced Bounded
Fun. Define $e_{\text{abbrev}}$ to be the term of Bounded Fun defined inductively as follows:

(i) If $e$ is a variable or constant, let $e_{\text{abbrev}} = e$.

(ii) $(ee')_{\text{abbrev}} = e'_{\text{abbrev}}$, if $e$ is of the form $\text{convert} \ [\sigma][\tau]$, 
\[ = (e_{\text{abbrev}} e'_{\text{abbrev}}), \text{otherwise}. \]

(iii) $(\text{fun} \ (x : \sigma), e)_{\text{abbrev}} = \text{fun} \ (x : \sigma), e_{\text{abbrev}}$.

(iv) $(e[\sigma])_{\text{abbrev}} = e_{\text{abbrev}}[\sigma]$.

(v) $(\text{all} \ (t \leq \sigma), e)_{\text{abbrev}} = \text{all} \ (t \leq \sigma), e_{\text{abbrev}}$.

(vi) $(\text{all} \ t, e)_{\text{abbrev}} = \text{all} \ t, e_{\text{abbrev}}$.

(vii) $(J_1 = e_1, \ldots, J_n = e_n)_{\text{abbrev}} = (J_1 = e_1_{\text{abbrev}}, \ldots, J_n = e_n_{\text{abbrev}})$.

(viii) $(e, J)_{\text{abbrev}} = e_{\text{abbrev}} \cdot J$.

Thus $e_{\text{abbrev}}$ is obtained from a translatable term $e$ by replacing all subterms of the form $\text{convert} \ [\sigma][\tau]$. We say that a term $e$ of Coerced Bounded Fun is a fattening of a term $e'$ of Bounded Fun if it is translatable and $e_{\text{abbrev}} = e'$.

We now show the close connection between derivations of terms in the Coerced and usual Bounded Fun.

4.3.4. LEMMA. Let $e$ be a translatable expression of Coerced Bounded Fun. Then, if $C, A \vdash_c e : \sigma$, one has $C, A \vdash_{\text{bf}} e_{\text{abbrev}} : \sigma$.

Proof. By induction on the complexity of the proof that $C, A \vdash_c e : \sigma$. Recall that the type inference rules of Bounded Fun contain all of those from Coerced Bounded Fun. Most of the proof is completely routine. The only interesting part is for the application rule. Suppose $C, A \vdash_c e : \sigma \rightarrow \tau$ and $C, A \vdash_c e' : \sigma$, yielding the result that $C, A \vdash (ee') : \tau$. Then by induction, $C, A \vdash_{\text{bf}} e_{\text{abbrev}} : \sigma \rightarrow \tau$ and $C, A \vdash_{\text{bf}} e'_{\text{abbrev}} : \sigma$.

(a) If $e$ is of the form $\text{convert} \ [\tau][\sigma]$, then $(ee')_{\text{abbrev}} = e'_{\text{abbrev}}$ and $C \vdash \sigma \leq \tau$. Thus by (sub), $C, A \vdash_{\text{bf}} e'_{\text{abbrev}} : \tau$, and $C, A \vdash_{\text{bf}} (ee')_{\text{abbrev}} : \tau$.

(b) Otherwise $(ee')_{\text{abbrev}} = (e_{\text{abbrev}} e'_{\text{abbrev}})$. Hence the result follows from the application rule.

Interestingly, one can go the other direction as well.

4.3.5. LEMMA. Let $e$ be a formula of Bounded Fun. If $C, A \vdash_{\text{bf}} e : \sigma$ then there is a translatable term $e'$ of Coerced Bounded Fun such that $e'$ is a fattening of $e$ and $C, A \vdash_c e' : \sigma$.

Proof. $e'$ is defined by induction on the length of the proof of $C, A \vdash_{\text{bf}} e : \sigma$. All steps but the one corresponding to the rule (sub) are trivial. Thus we present only the step for (sub) here. Suppose that $C, A \vdash_{\text{bf}} e : \sigma$ and hence by induction there is a translatable $e''$ such that
\textit{e''} is a fattening of \textit{e} and \( C, A \vdash \textsf{e'} : \sigma \). Suppose now that \( C \vdash \sigma \leq \tau \), and hence by (sub), \( C, A \vdash_{bf} \textsf{e} : \tau \). Then let \( \textsf{e}' = \texttt{convert} \, [\tau][\sigma] \textsf{e}'' \). Clearly \( \textsf{e}' \) is translatable, \( e'_{\text{abbrev}} = e''_{\text{abbrev}} = e \), and \( C, A \vdash_{c} \textsf{e}' : \tau \), as desired.

Our goal is to provide a meaning for a term \( e \) of Bounded Fun by first constructing a translatable term \( \textsf{e}' \) of Coerced Bounded Fun as in Lemma 4.3.5, and then giving \( e \) the same meaning as \( \textsf{e}' \). In order to show that this is well defined, we must show that all such terms of Coerced Bounded Fun have the same meaning. In fact, we will prove something stronger than this. In particular, if a term \( e \) of Bounded Fun can be given types \( \sigma \) and \( \tau \) where \( \sigma \leq \tau \), then the meaning in type \( \sigma \) can be coerced (using \texttt{convert}) into the meaning in type \( \tau \). In order to prove the key theorem we use the following lemma due to Curien and Ghelli.

\textbf{4.3.6. Lemma (Curien and Ghelli, 1990).} For all \( C, A, \) and \( e \in BF_{C,A} \) there is a (provably) minimum type \( \tau \) such that \( C, A \vdash_{bf} e : \tau \). I.e., if \( C, A \vdash_{bf} e : \tau' \), then \( C \vdash \tau \leq \tau' \).

The following theorem shows that all fattenings of a term \( e \in BF_{C,A} \) "cohere" nicely.

\textbf{4.3.7. Theorem.} For all \( C, A, \) and \( e \in BF_{C,A} \), if \( \tau \) is a minimum type for \( e \) with respect to \( C, A \), then there is a fattening \( \textsf{e}' \) of \( e \) such that

1. \( C, A \vdash_{c} \textsf{e}' : \tau \), and
2. For all fattenings \( e'' \) of \( e \), if \( C, A \vdash_{c} e'' : \tau' \), then
   \[ C, A \vdash_{c} \texttt{convert} \, [\tau'][\tau] \textsf{e}' = e''. \]

\textit{Proof.} The proof is by induction on the structure of terms. In view of Lemma 4.3.5 we only need to prove (2), with respect to the term \( \textsf{e}' \) in (1).

(i) \( e = x \). Let \( \textsf{e}' = x \) and let \( \tau \) be the minimum type for \( x \) as in Lemma 4.3.6. If \( e'' \) is a fattening of \( x \), then \( e'' = \texttt{convert} \, [\tau'][\tau] x \) for some \( \tau' \), and we are done. (Here and later, we implicitly use the transitivity of \texttt{convert}, i.e., rule (E2).)

(ii) \( e = c \), for \( c \) a constant. Similar to above.

(iii) \( e = (fk) \). Let \( \tau \) be the minimum type for \( e \), \( \rho \) be the minimum type for \( f \), and \( \pi \) the minimum type for \( k \), as in Lemma 4.3.6. By Lemma 4.3.5, the derivation that \( e \) has type \( \tau \) leads to a term \( \textsf{e}' \) of Coerced Bounded Fun, which is a fattening of \( e \) such that \( C, A \vdash_{c} \textsf{e}' : \tau \). By the definition of the fattening of a term, \( C, A \vdash_{c} \textsf{e}' = \texttt{convert} \, [\tau'] \texttt{convert} [\pi'][\pi] (f'' k'') \), where \( f'' \) and \( k'' \) are fattenings of \( f \) and \( k \), respectively. Since \( \tau \) is minimal, \( \tau' = \tau \) and \( C, A \vdash_{c} \textsf{e}' = (f'' k'') \) (by (E1) and \( \beta \)-reduction). Therefore \( C, A \vdash_{c} f'' : \pi' \to \tau \) and \( C, A \vdash_{c} k'' : \pi' \) for some \( \pi' \) such that \( C \vdash \pi \leq \pi' \).
(the inequality follows by the minimality of \( \pi \)). By induction, 
\[ C, A \vdash_c f'' = \text{convert} \left[ \pi' \to \tau \right][\rho] f' \] 
and 
\[ C, A \vdash_c k'' = \text{convert} \left[ \pi' \right][\pi] k' \]
for some \( f' \) and \( k' \) fattenings of \( f \) and \( k \), respectively. Therefore,
\[
C, A \vdash_c e' = (\text{convert} \left[ \pi' \to \tau \right][\rho] f')(\text{convert} \left[ \pi' \right][\pi] k') \\
= (\text{convert} \left[ \pi \to \tau \right][\rho] f')k' \quad \text{by (E6)},
\]
(\( \dagger \)).

Suppose \( e'' \) is a fattening of \( e \) such that 
\[ C, A \vdash_c e'' : \tau''. \]
Again by the definition of fattening, 
\[ C, A \vdash_c e'' = \text{convert} \left[ \tau'' \right][\tau'](f''k''), \]
where \( f'' \) and \( k'' \) are fattenings of \( f \) and \( k \), respectively. Let 
\[ C, A \vdash_c f'' : \pi'' \to \tau' \]
and 
\[ C, A \vdash_c k'' : \pi'' \]
for some \( \pi'' \) such that 
\[ C \vdash \pi \leq \pi'' \] (again the inequality follows by the minimality of \( \pi \)). Note that 
\[ C \vdash \tau \leq \tau' \] by Lemmas 4.3.4 and 4.3.6. By induction, 
\[ C, A \vdash_c f'' = \text{convert} \left[ \pi'' \to \tau' \right][\rho] f' \]
and 
\[ C, A \vdash_c k'' = \text{convert} \left[ \pi'' \right][\pi] k'. \]
Thus
\[
C, A \vdash_c (f''k'') = (\text{convert} \left[ \pi'' \to \tau' \right][\rho] f')(\text{convert} \left[ \pi'' \right][\pi] k') \\
= (\text{convert} \left[ \pi \to \tau' \right][\rho] f')k' \quad \text{by (E6)},
\]
(\( \dagger \)).

Therefore,
\[
C, A \vdash_c e'' = \text{convert} \left[ \tau'' \right][\tau'] \left(f''k'' \right) \\
= \text{convert} \left[ \tau'' \right][\tau'] \left(\text{convert} \left[ \tau' \right][\tau] e' \right) \quad \text{by above},
\]
(\( \dagger \)).

(iv) \( e = \text{fun} \ (x : \sigma). \ f \). Let \( \tau \) be the minimum type for \( e \), and let \( e' \) be the term of Coerced Bounded Fun which is a fattening of \( e \) and such that 
\[ C, A \vdash_c e' : \tau. \]
As in the previous case, 
\[ C, A \vdash_c e' = \text{convert} \left[ \tau \right][\tau'](\text{fun} \ (x : \sigma). \ f'), \]
where \( f' \) is a fattening of \( f \). As before \( \tau = \tau' \), so \( \tau = \sigma \to \pi \) for some \( \pi \), and 
\[ C, A \vdash_c e' = \text{fun} \ (x : \sigma). \ f'. \]
Note that \( \pi \) is minimal for \( f \) with respect to 
\[ C, A \cup \{ x : \sigma \} \] (otherwise \( \tau \) would not be the minimum type for \( e \)).

Suppose \( e'' \) is a fattening of \( e \) such that 
\[ C, A \vdash_c e'' : \tau''. \]
Again by the definition of fattening, 
\[ C, A \vdash_c e'' = \text{convert} \left[ \tau'' \right][\tau'](\text{fun} \ (x : \sigma). \ f''), \]
where \( f'' \) is a fattening of \( f \). By induction, 
\[ C, A \vdash_c f'' = \text{convert} \left[ \pi'' \right][\pi] f' \]
for some \( \pi'' \). Thus,
\[ C, A \vdash e'' = \text{convert} \left[ \tau'' \right]\left[ \tau' \right]\left( \text{fun} \left( x : \sigma \right), f'' \right) \]
\[ = \text{convert} \left[ \tau'' \right]\left[ \tau' \right]\left( \text{fun} \left( x : \sigma \right), \left( \text{convert} \left[ \pi'' \right]\left[ \pi \right] f' \right) \right) \]
\[ = \text{convert} \left[ \tau'' \right]\left[ \tau' \right] \left( \text{convert} \left[ \sigma \rightarrow \pi'' \right]\left[ \sigma \rightarrow \pi \right]\left( \text{fun} \left( x : \sigma \right), f' \right) \right) \]
\[ \quad \text{by (E3)} \]
\[ = \text{convert} \left[ \tau'' \right]\left[ \sigma \rightarrow \pi \right]\left( \text{fun} \left( x : \sigma \right), f' \right) \quad \text{by (E2)} \]

since \( \tau' \) must equal \( \sigma \rightarrow \pi'' \) for this to be typed,

\[ = \text{convert} \left[ \tau'' \right]\left[ \sigma \rightarrow \pi \right] e' \quad \text{by definition of } e'. \]

(v) \( e = f[\sigma] \). Let \( \tau \) be the minimum type for \( e \) and \( \rho \) be the minimum type for \( f \), as in Lemma 4.3.6. As before we get a term \( e' \) of Coerced Bounded Fun, which is a fattening of \( e \) and such that \( C, A \vdash e' : \tau \). By the definition of fattening, \( C, A \vdash e' = \text{convert} \left[ \tau \right]\left[ \tau' \right]\left( f''[\sigma] \right) \), where \( f'' \) is a fattening of \( f \). Since \( \tau \) is minimal, \( \tau' = \tau \) and \( C, A \vdash e' = f''[\sigma] \) (by (E1) and \( \beta \)-reduction). Here we get two cases, either (a) \( C, A \vdash f'' : \forall \alpha \leq \beta \cdot \pi \), or (b) \( C, A \vdash f'' : \forall \tau. \pi \) for some \( \beta, \pi \). We investigate these in turn.

(a) Suppose \( C, A \vdash f'' : \forall \alpha \leq \beta \cdot \pi \). Therefore \( \tau = \pi[\alpha/\iota] \). By induction, \( C, A \vdash f'' = \text{convert} \left[ \forall \alpha \leq \beta \cdot \pi \right]\left[ \rho \right] f' \), where \( f' \) is a fattening of \( f \). Therefore,

\[ C, A \vdash e' = \left( \text{convert} \left[ \forall \alpha \leq \beta \cdot \pi \right]\left[ \rho \right] f' \right)[\sigma]. \quad (\dagger\dagger) \]

Also \( C \vdash \sigma \leq \beta \), since the term is typeable.

Suppose \( e'' \) is a fattening of \( e \) such that \( C, A \vdash e'' : \tau'' \). Again by the definition of fattening, \( C, A \vdash e'' = \text{convert} \left[ \tau'' \right]\left[ \tau' \right]\left( f''[\sigma] \right) \), where \( f'' \) is a fattening of \( f \). Let \( C, A \vdash f'' : \forall \alpha \leq \beta' \cdot \pi' \). Thus \( C \vdash \sigma \leq \beta' \) and \( C \vdash \pi[\alpha/\iota] \leq \pi'[\alpha/\iota] \) (again the inequality follows by the minimality of \( \tau \)). Note that \( C \vdash \tau \leq \tau' \) by Lemmas 4.3.4 and 4.3.6. By induction, \( C, A \vdash f'' = \text{convert} \left[ \forall \alpha \leq \beta' \cdot \pi' \right]\left[ \rho \right] f' \). Thus,

\[ C, A \vdash f''[\sigma] = \left( \text{convert} \left[ \forall \alpha \leq \beta' \cdot \pi' \right]\left[ \rho \right] f' \right)[\sigma] \]
\[ = \text{convert} \left[ \pi[\alpha/\iota] \right]\left[ \pi[\alpha/\iota] \right]\left( \text{convert} \left[ \forall \alpha \leq \beta \cdot \pi \right]\left[ \rho \right] f' \right)[\sigma] \]
\[ \quad \text{by (E7)} \]
\[ = \text{convert} \left[ \pi[\alpha/\iota] \right]\left[ \pi[\alpha/\iota] \right] e'. \quad \text{by (\dagger\dagger)}. \]

The rest of this case is carried through exactly like case (iii).

(b) is similar to (a), using (E8) rather than (E7).

(vi)-(vii) on terms of the form all \( (t \leq \sigma). e \) and all \( t \cdot e \) are similar to (iv) using (E4) and (E5) rather than (E3).

(viii) \( e = f. I \). Let \( \tau \) be the minimum type for \( e \), \( \sigma \) be minimum type
for $f$, as in Lemma 4.3.6. As before we get a term $e'$ of Coerced Bounded Fun, which is a fattening of $e$ and such that $C, A \vdash e' : \tau$. As with previous cases, $C, A \vdash e' = f''$. If, where $f''$ is a fattening of $f$. Therefore $C, A \vdash f'' : (J_1 : \rho_1, ..., J_n : \rho_n)$, where $I = J_k$ and $\tau = \rho_k$ for some $k$. By induction, $C, A \vdash f'' = \text{convert} [(J_1 : \rho_1, ..., J_n : \rho_n)][\sigma] f'$ for some $f'$ a fattening of $f$. But note that by (E11) $C, A \vdash \text{convert} [(J_1 : \rho_1, ..., J_n : \rho_n)][\sigma] f'$. Therefore, $C, A \vdash f'' : (\sigma) f'$. Thus, $C, A \vdash f''$. Therefore, $C, A \vdash f''$. $C$

Suppose $e''$ is a fattening of $e$ such that $C, A \vdash e'' : \tau''$. As with previous cases, $C, A \vdash e'' = \text{convert} [\tau''][\tau'](f'', I)$, where $f''$ is a fattening of $f$. Suppose $C, A \vdash f'' : (J_1 : \rho_1'', ..., J_n : \rho_n'')$, where again $I = J_k''$ and $\tau' = \rho_k''$ for some $k$. By the minimality of $\tau$, $\tau \leq \tau'$. By induction, $C, A \vdash f'' = \text{convert} [(J_1 : \rho_1'', ..., J_n : \rho_n'')][\sigma] f''$. Thus,

$C, A \vdash f''$. $I$

$= (\text{convert} [(J_1 : \rho_1'', ..., J_n : \rho_n'')][\sigma] f')$. $I$

$= (\text{convert} [(\tau')][\tau'] f'). I$ by (E11),

$= (\text{convert} [(\tau')][(\text{convert} [(\tau) f'])]). I$ by (E2),

$= \text{convert} [(\tau')][(\text{convert} [(\tau) f'])] e'$. Therefore, $C, A \vdash f'' = \text{convert} [(\tau) f']$. $I$.

The proof is completed as in the previous cases.

(ix) $e = (J_1 = e_1, ..., J = e_n)$. This case is straightforward (using (E9)) and is omitted here. $\blacksquare$

4.3.8. Corollary. Let $e \in BF_{C, A}$.

(1) If $e_1$ and $e_2$ are fattenings of $e$ and $\tau$ is a type such that $C, A \vdash e_1 : \tau$, and $C, A \vdash e_2 : \tau$, then $C, A \vdash e_1 = e_2$.

(2) If $e_1$ and $e_2$ are fattenings of $e$ such that $C, A \vdash e_1 : \sigma$, and $C, A \vdash e_2 : \tau$, where $C \vdash \sigma \leq \tau$, then $C, A \vdash e_1 = e_2$.

Proof. (1) Let $\tau'$ be a minimal type for $e$ with respect to $C, A$ and let $e'$ be a fattening of $e$ guaranteed in Theorem 4.3.7. Therefore $C, A \vdash e_1 = \text{convert} [\tau'][\tau''] e'$ and $C, A \vdash e_2 = \text{convert} [\tau'][\tau''] e'$. Therefore, $C, A \vdash e_1 = e_2$.

(2) Similar. $\blacksquare$

We can now define the meaning of terms in Bounded Fun. Recall that a model $F$ of Coerced Bounded Fun is a model of Minimal Bounded Fun which has an interpretation for the constant convert which satisfies the axioms and rules in 4.3.1.
4.3.9. Definition. Let $F$ be a model of Coerced Bounded Fun. We define the interpretation of terms of Bounded Fun in $F$ as follows. Let $A$ be a syntactic type assignment and $C$ a collection of simple type constraints. If $\eta \models C, A$ then if $e \in BF_{C,A}$ and $C, A \vdash_{bf} e : \tau$, define $\llbracket C, A \vdash_{bf} e : \tau \rrbracket \eta = \llbracket C, A \vdash_{e} e' : \tau \rrbracket \eta$, where $e'$ is a fattening of $e$ such that $C, A \vdash_{e} e' : \tau$.

We must of course ensure that this definition makes sense, but this follows easily from the previous results in this section. By Lemma 4.3.5, for each $e$ as in the definition there is an appropriate translatable $e'$ in Coerced Bounded Fun. By Corollary 4.3.8, part 1, and the soundness of $F$ (as a model of Coerced Bounded Fun), it does not matter which such $e'$ we choose. This establishes the soundness of the subsumption rule and that the semantics of terms in (the original) Bounded Fun is well defined.

Note also that by Corollary 4.3.8, part 2, if $C, A \vdash_{e} e_1 : \sigma$, and $C, A \vdash_{e} e_2 : \tau$, where $C \vdash \sigma \leq \tau$, then $\llbracket C, A \vdash_{bf} e : \sigma \rrbracket \eta$ can be obtained by coercing $\llbracket C, A \vdash_{bf} e : \tau \rrbracket \eta$ from $\llbracket \sigma \rrbracket \eta$ to $\llbracket \tau \rrbracket \eta$ using the interpretation of convert.

In the next section we will construct an explicit model of Bounded Fun from partial equivalence relations. We will see that while the interpretation of convert will be a non-trivial polymorphic function, it will be defined in a very simple way which will enable us to verify (E1) through (E8) rather trivially.

5. THE MODEST MODEL OF BOUNDED FUN

5.1. Modest and $\omega$-Sets

In this section we show how to construct a model of Bounded Fun from a generalization of the category PER. This will be done by using some ideas of Eugenio Moggi, leading to the "small completeness" or closure under suitable products of the category PER. The approach in this subsection is more completely developed in Longo and Moggi (1988) (see also Rosolini, 1986; Hyland, 1988; Hyland et al. (1990); Hyland and Pitts, 1987; Carboni et al. 1987; Ehrhard, 1988; etc., for related category-theoretic work and Asperti and Longo, 1989, for the categorical background).

The point is that we need a frame (or "global" category) where also $T$, the collection of all types, is an object, so that we can interpret "universal quantification" over $T$. Recall, for this purpose, that these structures originated in (higher types or) generalized computability. An early and elegant approach was proposed by Malcev, in the early fifties, with the category EN of numbered sets, whose objects are pairs $(A, e_A)$, where $A$ is a countable set and $e_A : \omega \to A$ is an "enumeration" or total onto map.
Morphisms are functions \( f: A \to B \) such that, for some recursive \( f', f \circ e_A = e_B \circ f' \). Clearly any numbered set induces exactly one equivalence relation, e.g., on \( \omega \) (a total one!) by \( nAm \) iff \( e_A(n) = e_A(m) \), while lots of numbered sets induce the same e.r. (see the Out map below). The idea is to define a category which includes both PER and EN as full subcategories. For this, one may just take the category \( M \) in 5.1.1 of countable (modest) sets \( A \), with a partial enumeration \( f_A : \omega \to A \). As partial maps are just single-valued relations in \( \omega \times A \), we write \( n f_A(a) \) for \( f_A(n) = a \).

5.1.1. Definition. The category \( M \) (of modest sets) has as

objects. \( (A, f_A) \in M \) iff \( A \) is a set and \( f_A : \omega \to A \) is a partial onto map;

morphisms. \( f \in M[A, B] \) iff \( f : A \to B \) and \( \exists n, \forall a \in A, \forall p, f_A(a), (n \cdot p) f_B f(a) \).

\( M \) is not yet our “global” category. To define it, just drop the condition that the \( f_A \) relations are single-valued (we then call them “\( \leftarrow \_ \_ \_ \) relations or realizability relations, and we may omit the indices). That is, define:

5.1.2. Definition. The category \( \omega\text{-Set} \) has as

objects. \( (A, \leftarrow_A) \in \omega\text{-Set} \) iff \( A \) is a set, \( \leftarrow_A \subseteq \omega \times A \) and \( \forall a \in A, \exists p \leftarrow_A a \);

morphisms. \( f \in \omega\text{-Set} [(\forall, \leftarrow_A), (B, \leftarrow_B)] \) iff \( f : A \to B \) and \( \exists n, \forall a \in A, \forall p \leftarrow_A a, n \cdot p \leftarrow_B f(a) \) (notation. \( n \leftarrow_A \_ \_ \_ \_ \_ \rightarrow_B f) \).

It is obvious that \( M \) is a full subcategory of \( \omega\text{-Set} \). Moreover, PER may be fully and faithfully embedded in \( M \) (and \( \omega\text{-Set} \)). For every per, \( A \), define the \( \omega \)-set \( \text{In}(A) = (Q(A), e_A) \), where \( Q(A) \) is the set of equivalence classes of \( A \) and \( e_A \) is the usual membership relation restricted to \( \omega \times Q(A) \). Clearly, \( \text{In}(A) \) is a modest set, since \( A \) is a per and, hence, the elements of \( Q(A) \) are disjoint (nonempty) subsets of \( \omega \), i.e., \( e_A \) is single valued. Therefore, \( \text{In} \) defines an embedding from PER into \( M \). Conversely, for every modest set, \( (X, \leftarrow) \), we define a per \( \text{Out}(X) \), by

\[
n \text{Out}(X) m \iff \exists a \in X \text{ such that } n \leftarrow a \text{ and } m \leftarrow a.
\]

In conclusion, (\( \text{In}, \text{Out} \)) is an equivalence between the categories PER and \( M \), which extends to an isomorphism between pers and \( \omega \)-sets of the form \( (Q(A), e_A) \). Thus, even if \( M \) is not a small category, it is “essentially small” as it is equivalent to a small one. It is convenient to define it, as in categories one usually works ”up to isomorphisms.”

We are now going to use a strong closure property of the category \( M \) (and, by isomorphisms, of PER). As already mentioned, one crucial point
in the interpretation of Bounded Fun as a higher order language is the meaning of the universal quantifier. We are going to interpret it as an "indexed product" in the category $\omega$-Set. Recall that $\prod_{a \in A} g(a)$ is the set of functions, $f$, such that for all $a \in A$, $f(a) \in g(a)$.

5.1.3. **Definition.** Let $(A, \rightarrow_{\_}) \in \omega$-Set and $g: A \rightarrow \omega$-Set. Define the $\omega$-set $([\prod_{a \in A} g(a)], \rightarrow_{\prod_{\_} g})$ by

1. $f \in [\prod_{a \in A} g(a)]$ iff $f \in \prod_{a \in A} g(a)$ and $\exists n, \forall a \in A, \forall p \rightarrow_{\_} a, n \cdot p \rightarrow_{g(a)} f(a)$,

2. $n \rightarrow_{\prod_{\_} g} f$ iff $\forall a \in A, \forall p \rightarrow_{\_} a, n \cdot p \rightarrow_{g(a)} f(a)$.

Observe that if $g: A \rightarrow \omega$-Set is a constant function, $g(a) = (B, \rightarrow_{B})$ for all $a \in A$, say, then $([\prod_{a \in A} g(a)], \rightarrow_{\prod_{\_} g}) - (B^A, \rightarrow_{A \times B})$, the object representing $\omega$-Set $[A, B]$ in $\omega$-Set, where $n \rightarrow_{A \times B} f$ iff $\forall a \in A, \forall p \rightarrow_{\_} a, n \cdot p \rightarrow_{B} f(a)$. Indeed, $M$ and $\omega$-Set are CCCs.

One can directly obtain a product in PER, when the range of $g$ is restricted to $M$. (Note that this restriction is needed in order to obtain a well-defined per.)

5.1.4. **Definition.** Let $(A, \rightarrow_{\_}) \in \omega$-Set and $g: A \rightarrow M$. Let $[\prod_{a \in A} g(a)]_{\text{PER}} \in \text{PER}$ be defined by

$n \left[ \prod_{a \in A} g(a) \right]_{\text{PER}} m$ iff $\forall a \in A, \forall p, q \rightarrow_{\_} a, n \cdot p(\text{Out}(g(a))) m \cdot q$.

The products defined in 5.1.3 and 5.1.4 are isomorphic for $g: A \rightarrow M$.

5.1.5. **Theorem.** Let $(A, \rightarrow_{\_}) \in \omega$-Set and $g: A \rightarrow M$. Then $([\prod_{a \in A} g(a)], \rightarrow_{\prod_{\_} g})$ is in $M$ and is isomorphic to $\text{In}(\prod_{a \in A} g(a)]_{\text{PER}})$.

**Proof.** Let $\rightarrow_{\prod_{\_} g}$ be defined as in 5.1.3. We first prove that $\rightarrow_{\prod_{\_} g}$ is a single-valued relation. Assume that $n \rightarrow_{\prod_{\_} g} f$ and $n \rightarrow_{\prod_{\_} g} h$. We show that $\forall a \in A, f(a) = h(a)$ and, thus, that $f = h$. By definition $\forall a \in A, \forall p \rightarrow_{\_} a, n \cdot p \rightarrow_{g(a)} f(a)$ and $n \cdot p \rightarrow_{g(a)} h(a)$, and thus $f(a) = h(a)$ since, for all $a$, $\rightarrow_{g(a)}$ is single valued (and any $a$ in $A$ is realized by some natural number). The isomorphism is given by $J(f) = \{n | n \rightarrow_{\prod_{\_} g} f\}$; thus the range of $J$ is a collection of disjoint sets in $\omega$ (equivalence classes). $J$ and its inverse are realized by the (indices for the) identity function.

In conclusion, a product of pers indexed by an arbitrary $\omega$-set $A$, in the sense of 5.1.3, when $g: A \rightarrow M$, is an object of PER, to within isomorphism.

As should be clear by now, types are interpreted by the objects of PER, or, roughly, $T$ is interpreted by PER. We take a further step though, and
interact \( T \) as an object of \( \omega\text{-Set} \), by turning the entire category \( \text{PER} \) into an \( \omega \)-set. Indeed (the collection of objects of) \( \text{PER} \) is a set. Consider then

\[
\text{M}_\theta = (\text{PER}, \mapsto_{\text{PER}}) \in \omega\text{-Set}, \quad \text{where} \quad \mapsto_{\text{PER}} = \omega \times \text{PER},
\]
i.e., \( \forall n, \forall A \in \text{PER}, \, n \mapsto_{\text{PER}} A \). Thus we look at \( \text{PER} \) as a full subcategory of \( \omega\text{-Set} \), via \( \text{In} \), and as an object too. Clearly, \( \text{M}_\theta \) is an \( \omega \)-set, but it is not modest (\( \text{PER} \) could not be turned into a modest set, for cardinality reasons). Indeed, we never required \( T \) to be a type.

Thus, by 5.1.5, \( \text{PER} \) is equivalent to the subcategory \( \text{M} \) of \( \omega\text{-Set} \) closed by products indexed by any object in \( \omega\text{-Set} \), including, of course, the object \( \text{M}_\theta \) representing \( \text{PER} \) itself. This strong closure property of \( \text{PER} \) will give the mathematical meaning over Kleene’s \((\omega, \cdot)\) of the impredicative definition of second order types. (To be precise, though, more non-trivial work needs to be done. Namely, one has to prove that \( \text{M}_\theta \) is the object component of an internal category of \( \omega\text{-Set} \) and the product is the right adjoint of the diagonal functor, which makes all of this categorically sound, see Longo and Moggi, 1988 and Asperti and Longo, 1989).

### 5.2. Subtypes and Inheritance

We are now in the position to investigate subtypes in \( \text{PER} \). As before, \( \omega\text{-Set} \) is used as an (essential) tool.

#### 5.2.1. Definition.

Let \( (A, \mapsto_A), (B, \mapsto_B) \in \omega\text{-Set} \). Define

\[
(A, \mapsto_A) \leq (B, \mapsto_B) \iff \forall a \in A, \exists b \in B, \, \forall n (n \mapsto_A a \Rightarrow n \mapsto_B b).
\]

Let \( i: A \rightarrow B \) be a choice function such that if \( i(a) = b \) then \( \forall n (n \mapsto_A a \Rightarrow n \mapsto_B b) \). (Note that as there is not necessarily a unique such \( b \) for every \( a \), \( i \) has to choose one.)

#### 5.2.2. Remark.

This can be seen to be a straightforward generalization of the definition of subtype given in Section 2 as follows. By the \((\text{In}, \text{Out})\) correspondence between \( \text{PER} \) and \( \text{M} \) given after Definition 5.1.2, one has, for \( (A, \mapsto_A), (B, \mapsto_B) \in \text{M} \),

\[
\text{Out}(A) \leq \text{Out}(B) \quad \text{in } \text{PER} \iff (A, \mapsto_A) \leq (B, \mapsto_B) \quad \text{in } \omega\text{-Set}.
\]

Conversely, for \( A, B \in \text{PER} \), \( A \leq B \) iff \( \forall n, m \, (nAm \Rightarrow nBm) \) iff \( \text{In}(A) \leq \text{In}(B) \), or, viewing \( A \) and \( B \) as sets of ordered pairs, \( \text{In}(A) \leq \text{In}(B) \) iff \( A \subseteq B \) (cf. Definition 3.3). Thus the definition of \( \leq \) is preserved in the correspondence between \( \text{PER} \) and \( \omega\text{-Set} \) via \( \text{M} \). Moreover, take \( A, B \in \text{PER} \) such that \( A \leq B \), then the translation to \( \omega \)-sets results in \( i: Q(A) \rightarrow Q(B) \) such that \( i(n^A) = n^B \). In this case, it is easy to see that \( i \) is uniquely defined. Note that \( i \), the coercion morphism, is computed by all the indices of the identity function.
In the sequel, for technical convenience, we prefer to work in \( \text{PER} \), rather than \( M \), when unambiguous. For example, instead of a map \( g: A \to M \) we directly consider the map \( G = \text{Out} \circ g: A \to \text{PER} \), if helpful, or, also, we identify, \( \text{PER} \) with \( \text{In(\text{PER})} \). By this we avoid too many \( \text{In's} \) and \( \text{Out's} \).

5.2.3. \textbf{THEOREM.} Let \((A, \equiv_A), (A', \equiv_{A'})\) \( \in \omega\)-\textbf{Set} and let \( G: A \to \text{PER}, G': A' \to \text{PER} \). Assume that \( A' \leq A \) via \( i: A' \to A \) and that \( \forall a' \in A', G(i(a')) \leq G'(a') \). Then

\[
\left[ \prod_{a \in A} G(a) \right]_{\text{PER}} \leq \left[ \prod_{a' \in A'} G'(a') \right]_{\text{PER}}.
\]

\textbf{Proof.} Assume that \( n[\prod_{a \in A} G(a)]_{\text{PER}} m \) or, equivalently, that

\[
\forall a \in A, \forall p, q \equiv_A a, \quad (n \cdot p) G(a)(m \cdot q) . \tag{*}
\]

We need to prove that \( \forall a' \in A', \forall p, q \equiv_{A'} a', \quad (n \cdot p) G'(a')(m \cdot q) \). Observe that, by the definition of \( A' \leq A \) via \( i \), one has

\[
\forall a' \in A', \quad p \equiv_A a' \Rightarrow p \equiv_{A'} i(a'). \tag{\dagger}
\]

Thus, by (\dagger), (*), and the hypothesis on \( G \) and \( G' \),

\[
\begin{align*}
a' &\in A' \text{ and } p, q \equiv_A a' \\
&\Rightarrow p, q \equiv_{A'} i(a') \\
&\Rightarrow (n \cdot p) G(i(a'))(m \cdot q) \\
&\Rightarrow (n \cdot p) G'(a')(m \cdot q).
\end{align*}
\]

The simplicity of 5.2.3 is due to the merits of the "set-theoretic flavor" of the model. However, it is an important "structural" result of the present paper, as it shows that very general products preserve subtypes. For example, the usual inclusion of records, formalized in Bounded Fun, is realized in this model by 5.2.3, by taking \( A \) and \( A' \) in \( \text{PER} \) (as in Theorem 3.4(ii)). Similarly for general first- and second-order bounded quantifications, which all turn out to be handled similarly, i.e., by the same notion of product (indexed by different objects). In the next subsection, we use this property to construct a model for Bounded Fun.

5.3. Construction of the Model

Using the properties developed in the previous two sections, it is relatively straightforward (though a bit lengthy) to show how to construct a model of Extended Bounded Fun. Kinds are interpreted as elements of \( \omega\)-\textbf{Set} (in particular, \( T \) by \( M_0 = (\text{PER}, \equiv_{\text{PER}}) \)). The interpretation of
**MODEST MODEL OF RECORDS** 229

convert will be a function coerce, such that for $A \leq B$, one has coerce B $A([\{n\}_{\alpha}, n]_{B}) = [n]_{B}$ for $n A n$, as already pointed out. The effect of coerce B $A$ is simply to forget some structural information (captured in the partial equivalence relation $A$), which is no longer relevant in the supertype B. Indeed, the per $A$ is "finer" than $B$ (in the case of record types, this gives meaning to "$A$ has more fields than $B$").

We proceed by identifying the parts of the model according to the definitions given in Section 4.2. For the sake of simplicity, we keep considering PER both as a category (indeed, a full subcategory of $\omega$-Set) and as a set (indeed, the set component in the $\omega$-set, $M_0 = (\text{PER}, \vdash_{\text{PER}})$).

1. Let $\text{Type} = \text{PER}$ and $T = M_0 = (\text{PER}, \vdash_{\text{PER}})$, where $\vdash_{\text{PER}} = \omega \times \text{Type}$.

2. Define $\leq$ on $\text{Type}$ by $A \leq B$ iff $A \subseteq B$ (when looked at as a set of ordered pairs), as in Section 3.

3. For $B$ a per, let $\text{Type}^{\leq B} = \{A \in \text{Type} \mid A \leq B\}$ and $T^{\leq B} = (\text{Type}^{\leq B}, \vdash_{\text{Type}^{\leq B}})$, where for all $A \leq B$ and for all $n, n \vdash_{\text{Type}^{\leq B}} A$.

4. For $s \subseteq \omega$, let $D_s = (s, \vdash_s)$, where for $n \in s$, $\{m \mid m \vdash_s n\} = \{n\}$. $D_s$ is an $\omega$-set corresponding to the discrete per on $s$.

5. Define $\text{Kinds}$ to be the least subset of $\omega$-set containing $\{T\} \cup \{T^{\leq B} \mid B$ a per$\} \cup \{D_s \mid s \subseteq \omega\}$ and closed under products over $T$ (i.e., if $f: \text{Type} \to \text{Kinds}$, then $([\prod_{a \in T} f(a)], \vdash_{\prod_{a \in T} f(a)}) \in \text{Kinds}$) and function spaces (i.e., if $K, K' \in \text{Kinds}$ then so is $(\omega$-Set$[K, K'], \vdash_{K \to K'})$).

6. a. Let $I_{\text{Kind}}(T) = T$ and $I_{\text{Kind}}(D_s) = D_s$.

b. If $K = (K, \vdash)$ is a kind, then let $\text{Kind}^K = K$ (e.g., $\text{Kind}^\text{Type} = \text{Type}$).

c. Let the interpretation of $\Rightarrow$ be $\Rightarrow$, where $(K \Rightarrow K') = (\omega$-Set$[K, K'], \vdash_{K \Rightarrow K'})$.

d. Similarly, for $f: \text{Type} \to \text{Kinds}$, the interpretation of $\prod_T f$ is $\prod_T f = (\prod_{a \in T} f(a)), \vdash_{\prod_{a \in T} f(a)}$.

e. $\Phi_{K, K}$ is the identity on $\text{Kind}^{K \Rightarrow K'}$. Similarly, $\Phi_f$ is the identity on $\text{Kind}^{\prod_T f}$.

f. Let the interpretation of $\leq$ be $\subseteq$, where $(K \subseteq K') = (\omega$-Set$[K, K'], \vdash_{K \subseteq K'})$.

g. For each $s \subseteq \omega$, let $I_{\text{Kind}}(R_s) = \Pi_{s}$, where if $F \in \text{Kind}^{D_s \Rightarrow \text{Type}}$, $\Pi_s(F) = [\prod_{a \in D_s} F(a)]_{\text{PER}}$.

h. $I_{\text{Kind}}(\Rightarrow)$ is defined on $\text{Type}$ as in Section 3.

i. $I_{\text{Kind}}(\forall) = \prod$, where if $F \in \omega$-Set$[T, T]$, $\prod(F) = [\prod_{t \in \text{Type}} F(t)]_{\text{PER}}$.

j. $I_{\text{Kind}}(\forall) = \prod_{\leq}$, where if $B \in \text{Type}$, $F \in \omega$-Set$[\text{Type}^{\leq B}, T]$, $\prod_{\leq}(B)(F) = [\prod_{t \in \text{Type}^{\leq B}} F(t)]_{\text{PER}}$. 
7. We can then show that \( \text{Kind} = \langle \text{Kinds}, \{ \text{Kind}^k | k \in \text{Kinds} \}, \{ \Phi_{k,k} | k, k' \in \text{Kinds} \}, \{ \Phi_f | f \in \text{Type} \to \text{Kinds} \}, I_{\text{Kind}} \rangle \) is a kind structure.

8. To define a frame, make the following definitions:
   a. For \( B \in \text{Type} \), let \( \text{Dom}^B = Q(B) \),
   b. \( \Phi_{A,B}(\{ n \}_{A \to B}) = f \in Q(A) \to Q(B) \), where \( f(\{ p \}_A) = \{ n \cdot p \}_B \),
   c. For \( F = \omega \cdot \text{Set}[T, T] \) and \( \prod (F) = [\prod_{t \in \text{Type}} F(t)]_{\text{PER}} \), let \( \Phi_F(\{ n \}_{\prod F}) = g \in \prod_{t \in \text{Type}} Q(F(t)) \), such that for all \( B \in \text{Type} \), \( g(B) = \{ n \cdot 0 \}_{F(B)} \).
   d. For \( F = \omega \cdot \text{Set}[T^<\omega, T] \) and \( \prod \leq (B)(F) = [\prod_{t \in \text{Type} < s} F(t)]_{\text{PER}} \), let \( \Phi_{F,B}(\{ n \}_{\prod < (B)(F)}) = g \in \prod_{t \in \text{Type} < s} Q(F(t)) \), such that, for \( C \leq B \), \( g(C) = \{ n \cdot 0 \}_{F(C)} \).

(Note. The choice of "\( n \cdot 0 \)" in points c and d above is justified by 5.3.1 below.)

Note that by Theorem 5.2.3, the inequality on types satisfies all of the properties required for a kind structure in Definition 4.2.1. It can be shown that this frame is indeed a model of Minimal Bounded Fun by showing that all terms have an interpretation in the model. This proof is based on the fact that the partial recursive functions are represented in \( (\omega, \cdot) \), by the indices. Indeed, nothing else is required for the entire model construction. Thus, all above (and below) can be proved starting with any (possibly partial) combinatory algebra or model of Curry's combinators \( k \) and \( s \). (Besides their computational power, \( k \) and \( s \) explicitly appear in 5.3.1 (and in Longo and Moggi, 1988)). In a sense, \( (\omega, \cdot) \) is the "least" applicative structure one may start with, since any \( (p)CA \) contains (a representation of) the natural numbers.

This structure is also a model of Extended Bounded Fun where the interpretation of \text{convert} is given by an equivalence class (with respect to type \( [\forall t. \forall s \leq t. s \to t] \eta \) for any legal environment, \( \eta \)) which contains the number \( p \) with the property that for all \( m, n, p \cdot m \cdot n \) gives the index of an identity function on the natural numbers. (In terms of Curry's combinator, \( k \), take \( p = k \cdot (k \cdot i) \), where \( i \) is an index of the identity function.) The equivalence class of any such \( p \) will do. (The reader may wish to test his/her understanding of the above model and Definition 4.2.3 by verifying that any such \( p \) gives an element with the right properties.) By the above definition, if \( A \leq B \) then \( [\text{convert} t s] \eta[B/t, A/s] = \{ i \}_{A \to B} \) for \( i \) an index of the identity function. Thus the interpretation of \text{convert} \( t s \) does not change the representative of the partial equivalence class to which it is applied, only the partial equivalence relation with respect to which its equivalence class is formed. As a result it is completely trivial to verify that the axioms and rules, (E1) through (E8), of Explicit Bounded Fun are sound in this model.
Thus by Definition 4.3.9 and the remarks that follow it, the structure defined above is a model of Bounded Fun. Note that the function $\text{coerce}_{A,B}$ introduced in Section 2 is simply $[\text{convert } t/s] \eta[B/t, A/s]$.

We note here that, using the machinery developed above on general products, it would be rather simple to extend the model to dependent products. That is, we could interpret types of the form $\forall x : \sigma. \tau$, where $\sigma$ and $\tau$ are type expressions. These would be interpreted as sets of the form $[\prod_{a \in A} G(a)]_{\text{PER}}$, for $A$ the interpretation of $\sigma$, and $G$ the interpretation of $\lambda x : \sigma. \tau$, following 5.1.4. Of course, the syntax should include type constraint and type assignment rules to take care of these new constructs. This requires some work only because, when types may contain terms, the equational theory of types must be thoroughly described, as in Martin-Löf's approaches (see Coquand and Huet, 1985; Hyland and Pitts, 1987, Longo and Moggi, 1988 for the blending of first- and second-order in impredicative approaches).

We may finally relate the construction above to the interpretation of types suggested by Girard (1972) and Troelstra (1973) for system $\mathcal{F}$ and $\mathcal{II}$ order arithmetic. Following their work, we have interpreted types as partial equivalence relations, i.e., as objects of $\text{PER}$. Moreover, we interpreted $T$ by $M_0 = (\text{PER}, -_{\text{PER}})$ in $\omega\text{-Set}$, in order to have a “frame” where also the second order “$\forall$” could be understood as a product.

For the first-order predicative case or dependent types, it is easy to check that Troelstra's interpretation of types coincides with ours (cf. 5.1.4 and Troelstra, 1973, Section 4). As for the more challenging case, i.e., when quantification is over $T$, Girard and Troelstra suggested that $\forall t : T. \tau$ be interpreted as $n(\forall t : T. \eta) m$ for all $A \in \text{PER}$, $n(\exists t \eta[A/t]) m$.

This interpretation was proved to be sound by interpreting typed terms after erasing all of the type information from them. Surprisingly enough, Moggi (1986) hinted that the intersection over $\text{PER}$ is indeed a product, within Hyland's Effective Topos. Proofs, in various settings, were then suggested by Rosolini and Scott, Hyland and Freyd, Curien and Longo (see Rosolini, 1986; Hyland, 1988; Longo, 1988). Theorem 53.1 below says that the intersection is isomorphic to a product when working in $\omega\text{-Set}$ as the frame category.

5.3.1. Theorem. Let $(A, \mapsto_A) \in \omega\text{-Set}$ be such that $\mapsto_A = \omega \times A$ and let $G : A \to \text{PER}$. Then $[\prod_{a \in A} G(a)]_{\text{PER}} \cong \bigcap_{a \in A} G(a)$ in $\text{PER}$. 
Proof (Longo and Moggi, 1988). Set $S = \bigcap_{a \in A} G(a)$ and $\prod = \prod_{a \in A} G(a)_{\text{PER}}$. Define $Iso \in \text{PER}[S, \prod]$ as follows: Let $k$ be such that, for all $n, m$, $k \cdot n \cdot m = n$. Then, for $n \in \text{dom}(S)$, define

$$Iso(\{n\}_{S}) = \{k \cdot n\}_{\prod}.$$ 

Notice that $k \cdot n \in \text{dom}(\prod)$, since $\forall a \in A, nG(a)n$ and, hence,

$$\forall a \in A, \forall m, q \in \omega, \quad k \cdot n \cdot m \ G(a) \ k \cdot n \cdot q,$$

It is easy to see that the value of $Iso(c)$ is independent of the representative of the equivalence class. Clearly, $k \leftarrow Iso$.

To show $Iso$ is injective, suppose that $m, n \in \text{dom}(S)$ and that $\{n\}_{S} \neq \{m\}_{S}$. Therefore there is an $a \in A$ such that $m G(a) n$. Therefore, $k \cdot n \cdot 0 \ G(a) \ k \cdot m \cdot 0$, and thus $m G(a) _{\prod} k \cdot m$. Hence $Iso(\{n\}_{S}) \neq Iso(\{m\}_{S})$, so $Iso$ is injective.

It only remains to show that $Iso$ is surjective and that its inverse is realized by some natural number. Let $m \prod m$. By definition,

$$\forall a \in A, \forall u, q \in \omega, \quad m \cdot u \ G(a) \ m \cdot q, \quad \text{since} \quad \rightarrow_{A} = \omega \times A. \quad (*)$$

Take $n = m \cdot 0$. Then $\forall a \in A, n G(a)n$. We claim that $Iso(\{n\}_{S}) = \{k \cdot n\}_{\prod} = \{m\}_{\prod}$. For this, it is sufficient to show that $m \prod (k \cdot n)$. But by $(*)$, one has

$$\forall a \in A, \forall u, m \cdot u \ G(a) \ m \cdot 0.$$ 

Thus, since for all $q$, $k \cdot n \cdot q = n = m \cdot 0$,

$$\forall a \in A, \forall u, q, \quad m \cdot u \ G(a) \ k \cdot n \cdot q,$$

so $m \prod (k \cdot n)$. Finally, take $p$ such that for all $m, p \cdot m = m \cdot 0$. Clearly $p$ realizes $Iso^{-1}$.

Notice that in the proof above for $(A, \rightarrow_{A}) = M_{a}$, if $c = \{n\}_{S}$, then $\Phi_{c}(Iso(c))(a) = \{n\}_{G(a)}$, see also $8c$ above. Indeed, when universally quantified types are interpreted as intersections, elements of these types are equivalence classes in the intersections, such as $c$. Then $\Phi_{c}(Iso(c))(a)$ tells us how to apply $c$ to (the interpretation of) a type $a$ (and suggested by $8c$ and $d$ above). In conclusion, Theorem 5.3.1 is needed if one wants to show that the Girard–Troelstra interpretation of second-order types as intersection yields a model in the sense above (or as in Seely, 1987; see Hyland, 1988; Asperti and Longo, 1989; Meseguer, 1989). Indeed, it shows that their suggested model may be turned into a satisfactory interpretation with no need to erase types from terms. However, while types have the same interpretation, terms need to be interpreted differently, thus preserving their intended meaning as (typed) polymorphic programs.
Remark. In the notation above, $\Phi_G(Iso(c))((a) = \{n\}_{G(a)}$ is the definition of "polymorphic" application in Moggi's electronic mail message of February 1986, which suggested the various proofs of this simple but surprising fact. The result is relevant also for two more reasons that we have no space to discuss here. First, it gives an immediate understanding on how, under the assumptions in 5.3.1, the intersection of partial equivalence relations is (isomorphic to) their indexed product in Hyland's topos-theoretic model, $Eff$, of IZF (see Hyland, 1988; Longo and Moggi, 1988; Ehrhard, 1988; and Asperti and Longo, 1989, where the adjointness properties of this product are discussed). Second, it suggests an interesting foundational analysis, as the proof is based on the Uniformity Principle (UP, or the contrapositive of König's lemma), which is independent of IZF, but valid in $Eff$. The use of UP is implicit in the proof above, but the interested reader may recover it in the proof of the "surjectivity" of $Iso$ (see Rosolini, 1986 or Longo, 1988, for explicit discussions).

6. PROBLEMS WITH BOUNDED FUN AND FUTURE WORK

While the Modest model described above provides a sound interpretation of the language, the model indicates some difficulties with the language. The following lemma provides an indication that Bounded Fun is either too strong or too weak (it is debatable which it is) to express important operations that would be expected in an object-oriented language.

6.1. Lemma. Let $\sigma$ be a type expression and $\eta$ be an environment for the Modest model described above. Then $[\forall t \leq \sigma. t \rightarrow t] \eta$ contains only the interpretation of all $(t \leq \sigma). \lambda x : t. x$, the restriction of the polymorphic identity function.

Proof. Let $h$ be in the domain of the per, $[\sigma] \eta$. We claim that $h$ acts as the polymorphic identity for all types $\leq B$. Now let $m$ be in the dom($B$) and let $C_m = \{(m, m)\}$, representing the type with only one element, $\{m\}_B$. Thus $C_m \leq B$. By definition of the semantics of bounded quantification, $h \cdot n(C_m \rightarrow C_m)h \cdot n$ for all $n$ (actually, $8c-d$ gives $h \cdot 0(C_m \rightarrow C_m)h \cdot 0$; but this is the same by the definition of $\vdash \text{per}$ in $M_0$). Then, by the definition of $\rightarrow$ on pers, it follows that for all $p, q$, if $pC_m q$ then $(h \cdot n \cdot p) C_m (h \cdot n \cdot q)$. Since $C_m$ only contains $(m, m)$ this implies that $(h \cdot n \cdot m) C_m (h \cdot n \cdot m)$. But this can only happen if for all $n, h \cdot n \cdot m = m$.

Since we can repeat this for all $m \in \text{dom}(B)$, it follows that for all $n, h \cdot n$ is the identity function on $\text{dom}(B)$; namely, $h = k \cdot i$, for an index $i$ of the
identity function on \( \text{dom}(B) \). Thus for all \( C \leq B \), and for all \( n \),
\[
\{ h \cdot n \}_C \rightarrow C \equiv \{ i \}_C \rightarrow C
\]
represents the identity function on the type \( C \). Thus
\[
\{ h \}_A = \{ \text{all } (t \leq \sigma). \lambda x : t. x \}\eta \text{ since our model is extensional.}
\]

This lemma shows that the Modest model contains no functions which can be used for polymorphic record updates. That is, suppose we wish to define:

\[
\text{simple-update}_I : \forall t \leq (I : \text{Integer}). t \rightarrow t
\]

which is intended to apply some uniform (polymorphic) operation to record component \( I \), while leaving the other portions of the record alone. Unfortunately, by the above lemma, the only such operation which is in the model (and hence is definable) is the (bounded) polymorphic identity function. (On the other hand, there are many functions with type
\[
\forall t \leq (I : \text{Integer}). t \rightarrow (I : \text{Integer}).
\]

We can try to provide a somewhat more complex solution by deciding that rather than depending on a fixed update function, we might better supply the \( \text{update}_I \) function with another (polymorphic) function which is defined only on subtypes of the type of the \( I \)-component. For example:

\[
\text{update}_I : \forall s. \forall t \leq \{ I : s \}. (\forall u \leq s. u \rightarrow u) \rightarrow (t \rightarrow t).
\]

Unfortunately, we run into the same problem here, since the type
\[
(\forall u \leq s. u \rightarrow u)
\]
contains only the (bounded) polymorphic identity function.

It seems that the best we could do is to extend the type expressions by adding the expression \( t. I \) if \( t \leq \{ I : s \} \) for some type \( s \). This expression would denote the type of the \( I \)-component of the type \( t \). We could then write:

\[
\text{weaker-update}_I : \forall s. \forall t \leq \{ I : s \}. (t. I \rightarrow t. I) \rightarrow (t \rightarrow t).
\]

Thus \( \text{weaker-update}_I \) takes a function which takes an argument of the same type as that of the \( I \)-component of and returns an element of the same type. However, such a term does not provide as much parametric polymorphism since the functions supplied which operate on the various \( t. I \)'s need not have any connection to any other.

The fact that this model does not contain many polymorphic functions (at least in the types of the form \( \forall t \leq \sigma. t \rightarrow t \)), points out serious weaknesses in the expressibility of Bounded Fun. Since our model is sound, any term of these types which was expressible would have to be represented in the model. Since they are not represented, they must not be expressible.

What is the problem? In a sense there are too many subtypes. The key to the proof of the above lemma was the fact that for every element of a type, that type has a subtype which contains only that element. As a result
since the polymorphic functions were required to take elements of any subtype back to that subtype, all elements had to be fixed, or in other words the polymorphic function had to behave as the identity.

A possible solution to this problem is to restrict the notion of subtype to only "object-oriented" subtype. After all, it is the "subset" types which seem to be causing us problems. While such a restriction might rule out subtypes such as char \leq string, integer \leq real, etc., it is not entirely clear that these are desirable subtypes. Efforts are underway by several researchers to revise the language and create richer models (e.g., see Cardelli and Longo, 1989).

A promising alternative approach is to add a new construct to the language denoting extensions to record types. See Wand (1989) and Jategaonkar and Mitchell (1988) for variants on this approach.

There is another more radical approach to the problem, which is to separate the notions of subtype and inheritance. Several authors (e.g., Liskov, 1988; Snyder, 1986) have argued recently that object-oriented programming involves (at least) two quite different notions: code reuse, and inheritance of representation. Code reuse depends on having functions of the same name and the same (parameterized) functionality in various types. For instance, the code from a quicksort can be applied to any type which supports a binary Boolean-valued operation on the type. (Of course, whether it behaves properly or even terminates depends on the meaning of these operations.) Thus code reuse (at least in the sense of object-oriented programming) depends only on the interface of the operations associated with the type (i.e., the signatures of the operations defined on the type). The semantics of these operations is irrelevant for type-checking. On the other hand, many uses of object-oriented programming depend on objects of a subtype inheriting the methods of a supertype (although it is not atypical to redefine the inherited operations).

In order to support this view of object-oriented programming, we would expect to throw away rule (sub), for type checking (the authors have always been uncomfortable with this rule anyway). When looking at types or modules, it is necessary only to look at the signature of the operations supported. We would then define \( \sigma \leq \tau \) iff \( \sigma \) supports the same (names of) operations as \( \tau \). However, objects of type \( \sigma \) cannot be treated as objects of type \( \tau \). Subtyping thus becomes purely syntactic, having no semantic consequences.

The notions of subtype and inheritance supported by most object-oriented languages include a combination of these syntactic and semantic properties. Elements of subtypes inherit operations whenever possible and convenient, but languages also allow the user to redefine operations whenever desired (without affecting the relation of subtype). In these languages (e.g., Smalltalk), the representation of a subtype is inherited from the supertype, but there is no compelling reason for this. The abstract types of
Smalltalk fit into this view quite easily. These ideas remain to be worked out in detail, but we have hopes of providing a complete semantic specification of a language supporting these ideas.

7. Summary and Relation to Other Work

In this paper we have given a formal semantics for the language Bounded Fun from CW, (1985), which supports both parametric and subtype polymorphism. (Subtype polymorphism is based on inheritance and might also be called structural polymorphism.) We have also shown how to use partial equivalence relations to model inheritance in this language, which supports the notion of subtype and record types. (Our language actually differs from theirs in some minor ways.) A generalization of partial equivalence relations, known as $\omega$-sets, were used in combination with modest sets (pers) to provide the first known model of Bounded Fun (with explicit polymorphism). The connections with previous work on the semantics of explicit parametric polymorphism, based on the Girard–Troelstra interpretation (e.g., Mitchell, 1986), is given by noting that the semantics of polymorphic types presented here (via dependent products) is isomorphic to that given by the intersection interpretation of polymorphism.

Bainbridge et al. (1990) introduced the subcategory of PER, I, which contains the same objects as PER, but whose morphisms consist only of those from PER which are witnessed by an index of the identity function. These morphisms consist of exactly those functions of the form $\text{coerce } BA$, where $A \leq B$ according to the definition in this paper. They show that every morphism in PER can be decomposed into an isomorphism followed by a morphism from I followed by another isomorphism, a very simple but surprising property of PER. The results in this paper were discovered independently of those results, which were not meant to understand subtyping and inheritance.

We note here that an alternative model, according to the definition in Bruce et al. (1990), could be constructed using a subcategory of $\omega$-sets, called multi-modest sets, MM. Its objects have the property that if $A$ is a multi-modest set and $a, b \in A$ such that $n \vdash_A a$ and $n \vdash_A b$, then $\{n\,|\,n \vdash_A a\} = \{n\,|\,n \vdash_A b\}$. Notice that object $M_0$ of $\omega$-Set, which internally represents PER and is given at the end of Section 5.1, is a multi-modest set. However, category-theoretical problems in embedding $M$ as an internal category and constructing a right adjoint of the diagonal function led us to drop MM in favor of $\omega$-Set. Indeed, while the $\omega$-Set construction leads to a model in the sense of Seely (1986) (see Longo and Moggi, 1988, and Asperti and Longo, 1989, for details), this does not seem possible when using MM.
An entirely different model of Bounded Fun, based on the "interval semantics," is given in Martini (1988). Martini's model does not interpret dependent types and, thus, uses a different method to support record types. It does, however, interpret recursive definitions of functions. Following our work, Amadio (1988) investigates a variant of the modest model which interprets both records and recursive definitions of functions.

An important issue is whether one can consistently extend Bounded Fun with "arbitrary" recursive definitions of data types (i.e., find a model for such extensions). Breazu-Tannen et al. (1989) have recently produced a technique for using arbitrary models of the second-order lambda calculus as models of Bounded Fun by encoding bounded quantification using explicit coercers. Their technique involves translating formulas of Bounded Fun into formulas of the second-order lambda calculus. Their translation is somewhat more complex than ours, resulting in translations of types as well as terms of Bounded Fun. However, since their translation allows them to interpret Bounded Fun in an arbitrary model of the second-order lambda calculus, they can interpret Bounded Fun in a model which provides solutions to recursive domain equations. We do not know how their translation into a second-order model based on PER relates to our model of Bounded Fun. As a matter of fact, further interesting variants of this model have been used in Amadio (1989) to interpret extensions of Bounded Fun which allow recursively defined types. Moreover, Amadio (1989) gives several informative ways to solve domain equations on his categories of pers and confirms by this the richness and flexibility of the PER based models (or, perhaps, the relevance of the underlying Effective Topos for understanding polymorphism and more).

Finally, we note with interest the recent paper of Canning et al. (1989) which introduces an extension of Bounded Fun in which the expression $\tau$ in the formula $\forall (t \leq \tau). e$ may contain $t$ as a free variable. As demonstrated in the cited paper, this provides added expressibility in the language, capturing more programs of interest to the object-oriented community. Interestingly, the PER semantics presented in this paper appear to also provide a sound semantic model for this new construct.

ACKNOWLEDGMENTS

Longo thanks his (former) students Eugenio Moggi, Andrea Asperti, and Roberto Amadio for several instructive discussions and for the recent joint work, as stimulating as the previous common experiences. With P. L. Curien, he had a chance to discuss Moggi's understanding of "products-as-intersections." Dana Scott convinced us in several lectures and discussions of the relevance of the various categories of Modest Sets for the purposes of semantics. Thanks also to Luca Cardelli and Peter Wegner for stimulating our interest in subtypes and inheritance. The lemma in Section 6 arose as a result of a discussion with John Mitchell at a
workshop in "Category Theory and Semantics" at Carnegie-Mellon University in the spring of 1988. Special thanks to Breazu-Tannen et al. (1989), who pointed out some serious omissions related to the coherence of terms in Coerced Bounded Fun in an earlier version of this paper.

Received January 25, 1989; final manuscript received December 4, 1989

References


