A semantic basis for Quest[1]

Luca Cardelli
Digital Equipment Corporation
Systems Research Center

Giuseppe Longo [2]
LIENS, Ecole Normale Supérieure, Paris

Abstract

Quest is a programming language based on impredicative type quantifiers and subtyping within a three-level structure of kinds, types and type operators, and values.

The semantics of Quest is rather challenging. In particular, difficulties arise when we try to model simultaneously features such as contravariant function spaces, record types, subtyping, recursive types, and fixpoints.

In this paper we describe in detail the type inference rules for Quest, and we give them meaning using a partial equivalence relation model of types. Subtyping is interpreted as in previous work by Bruce and Longo, but the interpretation of some aspects, namely subsumption, power kinds, and record subtyping, is novel. The latter is based on a new encoding of record types.

We concentrate on modeling quantifiers and subtyping; recursion is the subject of current work.

[1] Preprint. Appeared in J. of Functional Programming 1 (4), pp. 417-458, October 1991.

[2] This author's work has been supported in part by Digital Equipment Corporation.

Revised 3/15/90

Contents

1. Introduction 1
2. Quest rules 3
2.1 Terms 4
2.2 Judgments 5
2.3 Environments and variables 5
2.4 Equivalence and inclusion 6
2.5 Subsumption vs. coercion 7
2.6 Power kinds 8
2.7 Operator kinds 9
2.8 The kind of types 9
2.9 Formal system 11
2.10 Records and other encodings 17
3. PER and ω-Set 21
3.1 Semantics of kinds and types 22
3.2 Inclusion and power kinds 27
3.3 Operator kinds 29
3.4 The kind of types 30
3.5 Records 32
4. Semantic interpretation of Quest_c 33
4.1 Interpretation 34
4.2 Emulating coercions by bounded quantification 37
5. Semantic interpretation of Quest 38
5.1 Preliminaries and structures 40
5.2 Interpretation 42
6. Conclusions 46
Acknowledgements 47
References 49

1. Introduction

Type theory provides a general framework for studying many advanced programming features including polymorphism, abstract types, modules, and inheritance. (See [Cardelli Wegner 85] for a survey.) The Quest programming language [Cardelli 89] attempts to take advantage of this general framework to integrate such programming constructs into a flexible and consistent whole.

In this paper we focus on the Quest type system, by describing and modeling its most interesting features. At the core of this system is a three-level structure of kinds, types (and type operators), and values. Within this structure we accommodate impredicative type quantifiers and subtyping. Universal type quantifiers can then be used to model type operators, polymorphic functions, and ordinary higher-order functions. Existential type quantifiers can model abstract types. Subtyping supports (multiple) inheritance, and in combination with quantifiers results in bounded-polymorphic functions and partially abstract types. Subtyping is realized in a uniform way throughout the system via a notion of *power kind*, where P (A) is the kind of all subtypes of A.

Formally, Quest is an extension of Girard's Fω [Girard 72] with additional kind structure, subtyping structure, recursive types, and fixpoints at all types. Alternatively, it is a higher-order extension of the calculus studied in [Curien Ghelli 90], which is the kernel of the calculus in [Cardelli Wegner 85]. Recursion is necessary to model programming activities adequately, and causes us to abandon the Curry-Howard isomorphism between formulas and types.

New kinds and types can be easily integrated into the basic Quest system to model various programming aspects. For example, basic types can be added to model primitive values and their relations [Mitchell 84]; record and variant types can be introduced to model object-oriented programming [Cardelli 88, Wand 89, Cardelli Mitchell 89, Cook Hill Canning 90]; and set types can be introduced to model relational data bases [Ohori 87]. In all these cases, subtyping performs a major role. Many of these additional type constructions can however be encoded in a very small core system, which is the one we investigate in this paper.

The type rules we consider are very powerful, but not particularly complex or unintuitive from a programming perspective. This contrasts with the semantics of Quest, which is rather challenging. In particular, difficulties arise when we try to model simultaneously features such as contravariant function spaces, record types, subtyping, recursive types, and fixpoints. In this paper we concentrate on modeling quantifiers and

subtyping; recursive types and values are an active subject of research [Amadio 89] [Abadi Plotkin 90] [Freyd Mulry Rosolini Scott 90].

The model we present for such advanced constructions is particularly simple; the basic concepts are built on top of elementary set and recursion theory. This model has been investigated recently within the context of Category Theory, in view of the relevance of Kleene's realizability interpretation for Category Theory and Logic. Our presentation applies and further develops, in plain terms and with no general categorical notions, the work carried on in [Longo Moggi 88] and [Bruce Longo 89]. Our work is also indebted to that by Amadio, Mitchell, Freyd, Rosolini, Scedrov, Luo and others (see references).

The presentation of the formal semantics is divided into two parts, corresponding to sections 4 and 5, where we discuss variants of the language with and without explicit coercions. However, the underlying mathematical structure is the same and the interpretations are strictly related.

We conclude this section with a few examples, both to introduce our notation and to provide some motivation.

The polymorphic identity function below introduces the universal quantifier over types (**P**) along with **l**-abstraction over types (**l**(X::TYPE)) and type application, and the function space operator () along with **l**-abstraction over values (**l**(x:X)) and value application:

```
let id : P(X::TYPE) (XX) =
    l(X::TYPE) l(x:X) x

id(Int)(3) = 3 : Int
```

Abstract types are obtained by existential quantification over types (**S**) [Mitchell Plotkin 85]. (As is well known, these existential quantifiers, with their associated primitives, can be defined in terms of **P** and . Similarly, cartesian product (), can be defined from .) The following might be the type of a package providing an abstract type X, a constant of type X, and an operation from X to Int:

```
S(X::TYPE) (X   (XInt))
```

Bounded universal quantifiers allow us to write functions that are polymorphic with respect to all the subtypes (<:) of a given type. This is particularly useful for subtypes of record types, which are generally meant to

model object types in object-oriented programming languages. Here ⟨age:Int⟩ is the type of records that contain a field age of type Int, and ⟨age=5, color=red⟩ is a value of type ⟨age:Int, color:Color⟩, which is a subtype of ⟨age:Int⟩. The following ageOf function computes the age of any member of a subtype of ⟨age:Int⟩.

let ageOf : Π(X<:⟨age:Int⟩) (XInt) =
  λ(X<:⟨age:Int⟩) λ(x:X) x.age

ageOf(⟨age:Int, color:Color⟩)(⟨age=5, color=red⟩) = 5 : Int

Bounded existential quantifiers are useful for representing types that are *partially abstract* in the sense that they are known to be subtypes of a given type, but are not completely specified:

Σ(X<:⟨age:Int⟩) ...

Bounded existential quantifiers also model types that are subtypes of abstract or partially abstract types:

Σ(X<:⟨age:Int⟩) Σ(Y<:X) ...

These last two features are present, in specific forms, in Modula-3 [Cardelli Donahue Glassman Jordan Kalsow Nelson 88].

We refer to [Cardelli 89] for detailed programming examples that use the full power of the system.

The paper is organized as follows. Section 2 describes the formal theory of Quest, including its typing rules, and can be understood on its own. Sections 3, 4, and 5 are more technical and are concerned with semantics. Section 3 provides background material on partial equivalence relation (p.e.r.) models, and more specific material on subtyping. Section 4 gives meaning to Quest_C (with explicit coercions), while section 5 gives meaning to Quest (with implicit subsumption).

2. Quest rules

In this section we discuss the typing and reduction rules for Quest. We use K,L,M for kinds; A,B,C for types and operators; a,b,c for values; X,Y,Z for type and operator variables; and x,y,z for value variables. We also use T for the kind of all types, and $P(B)$ for the kind of subtypes of B. In general, we

use capitalized names for kinds and types, and lower-case names for values.

2.1 Terms

The *pre-terms* are described by the following syntax. Only those pre-terms that are validated by the rules in the following subsections are legal *terms*.

K ::=		Kinds
	P (A)	the kind of all subtypes of a type
	P(X::K)L	the kind of operators between kinds
A ::=		Types and Operators
	X	type and operator variables
	Top	the supertype of all types
	P(X::K)B	polymorphic types
	A B	function spaces
	l(X::K)B	operators
	B(A)	operator application
	m(X)A	recursive types
a ::=		Values
	x	value variables
	top	the distinguished value of type Top
	l(X::K)b	polymorphic functions
	b(A)	polymorphic instantiation
	l(x:A)b	functions
	b(a)	function application
	$c_{A,B}$(a)	coercions
	m(x:A)a	recursive values

The following abbreviations will be used:

T	P (Top)	the kind of all types

P(X)L	P(X::T)L	P(X<:A)L	P(X::P (A))L
P(X)B	P(X::T)B	P(X<:A)B	P(X::P (A))B
l(X)B	l(X::T)B	l(X<:A)B	l(X::P (A))B
l(X)b	l(X::T)b	l(X<:A)b	l(X::P (A))b

From the abbreviations above we can see that this calculus includes all the

terms of Fω [Girard 72] and Fun [Cardelli Wegner 85].

2.2 Judgments

The formal rules are based on eight primitive judgment forms plus three derived ones, listed below.

⊢ E env	E is an environment
E ⊢ K kind	K is a kind (in an environment E)
E ⊢ A::K	type A has kind K
E ⊢ A type	A is a type (abbr. for E ⊢ A::T)
E ⊢ a:A	value a has type A
E ⊢ K<::L	kind K is a subkind of kind L
E ⊢ A<:B	type A is a subtype of type B (abbr. for E ⊢ A::P (B))
E ⊢ K<::>L	K and L are equivalent kinds
E ⊢ A<:>B::K	A and B are equivalent types or operators of kind K
E ⊢ A<:>B type	A and B are equivalent types (abbr. for E ⊢ A<:>B::T)
E ⊢ a≡b:A	a and b are equivalent values

A judgment like E ⊢ a:A is interpreted as defining a relation between environments, value terms, and type terms. This relation is defined inductively by axioms and inference rules, as described in the following sections. The rules are then summarized in section 2.9.

2.3 Environments and variables

An environment E is a finite sequence of type variables associated with kinds, and value variables associated with types. We use dom(E) for the set of type and value variables defined in an environment.

[Env ∅]
$$\frac{}{\vdash \emptyset \text{ env}}$$

[Env X]
$$\frac{E \vdash K \text{ kind} \quad X \notin dom(E)}{\vdash E, X::K \text{ env}}$$

[Env x]
$$\frac{E \vdash A \text{ type} \quad x \notin dom(E)}{\vdash E, x:A \text{ env}}$$

[Var X]
$$\frac{\vdash E', X::K, E'' \text{ env}}{E', X::K, E'' \vdash X :: K}$$

[Var x]
$$\frac{\vdash E', x:A, E'' \text{ env}}{E', x:A, E'' \vdash x : A}$$

2.4 Equivalence and inclusion

Equivalence of kinds (<::>) is the least congruence relation over the syntax of kinds that includes the following rule involving type equivalence:

[KEq P]
$$\frac{E \vdash A <:> A' \text{ type}}{E \vdash P(A) <::> P(A')}$$

Equivalence of types and operators (<:>) is the least congruence relation over the syntax of types that includes **b** and **h** type conversions (shown later), and the following rule for recursive types. Here AX means that A must be *contractive* in X in order to avoid non-well-founded recursions; see the definition in 2.9. The third rule below claims that every contractive context C has a unique fixpoint.

[TF m]
$$\frac{E, X::T \vdash A \text{ type} \quad AX}{E \vdash m(X)A \text{ type}}$$

[T m]
$$\frac{E, X::T \vdash A \text{ type} \quad AX}{E \vdash m(X)A <:> A\{X\,m(X)A\} \text{ type}}$$

[TEq Contract]
$$\frac{E \vdash A <:> C\{XA\} \text{ type} \quad E \vdash B <:> C\{XB\} \text{ type} \quad CX}{E \vdash A <:> B \text{ type}}$$

Inclusion of recursive types is given by the following rule, working inductively from the inclusion of the recursive variables to the inclusion of the recursive bodies:

[TIncl m]
$$\frac{E \vdash m(X)A \text{ type} \quad E \vdash m(Y)B \text{ type} \quad E, Y::T, X<:Y \vdash A <: B}{E \vdash m(X)A <: m(Y)B}$$

Equivalence of values (ÿ) is the least congruence relation over the syntax of values that includes **b** and **h** value conversions (shown later), together with the following rule for recursive values:

[m]
$$\frac{E \int m(x{:}A)b : A}{E \int m(x{:}A)b \ddot{y} b\{x m(x{:}A)b\} : A}$$

The rules for recursive types and values will not be modeled in the later sections. Nonetheless, we consider them an essential part of the language, and refer the reader to [Amadio 89], [Abadi Plotkin 90], and [Freyd Mulry Rosolini Scott 90] for related and ongoing work.

The following rules state that the property of having a kind (respectively a type) is invariant under kind (respectively type) equivalence; that is, equivalent kinds and types have the same extensions:

[KExt] (Kind Extension)
$$\frac{E \int A{::}K \quad E \int K <::> L}{E \int A :: L}$$

[TExt] (Type Extension)
$$\frac{E \int a{:}A \quad E \int A <:> B \text{ type}}{E \int a : B}$$

The relations of type and kind inclusion are reflexive and transitive:

[KIncl Refl]
$$\frac{E \int K <::> L}{E \int K <:: L}$$

[KIncl Trans]
$$\frac{E \int K <:: L \quad E \int L <:: M}{E \int K <:: M}$$

[TIncl Refl]
$$\frac{E \int A <:> B \text{ type}}{E \int A <: B}$$

[TIncl Trans]
$$\frac{E \int A <: B \quad E \int B <: C}{E \int A <: C}$$

We shall see shortly that the subtype relation is actually defined in terms of power kinds, then all the rules written in terms of subtyping are interpreted as rules about power kinds.

2.5 Subsumption vs. coercion

The following rules reflect the set-theoretical intuitions behind the subtyping relation. We present two alternatives: subsumption and coercion.

Subsumption formalizes a computationally natural way of looking at subtypes. When viewing computations as type-free activities, any element of a type is directly an element of its supertypes:

[TSub] (Subsumption)

$$\frac{E \int a:A \quad E \int A <: B}{E \int a : B}$$

A mathematical model of Quest with subsumption is given in part 5. That model is the main semantic novelty of this paper.

Before that, in part 4, we consider a system without subsumption, called $\text{Quest}_{\mathbf{c}}$. In $\text{Quest}_{\mathbf{c}}$, subsumption is replaced by a *coercion* rule, where a value of a type A must be explicitly injected into a supertype B by a coercion function $\mathbf{c}_{A,B}$. Invariance under type inclusion will be true only modulo coercions in the most straightforward semantics given in part 4.

[TSub] (Coercion)

$$\frac{E \int a:A \quad E \int A <: B}{E \int \mathbf{c}_{A,B}(a) : B}$$

In the semantics of $\text{Quest}_{\mathbf{c}}$ we obtain a single coercion function $\mathbf{c}: \mathbf{P}(X::\mathsf{T})\ \mathbf{P}(Y<:X)\ YX$; then $\mathbf{c}(B)(A)$ gives meaning to $\mathbf{c}_{A,B}$.

Coercions satisfy the following basic rules; more rules will be given later.

[VCoer Id / $\text{Quest}_{\mathbf{c}}$]

$$\frac{E \int a:A}{E \int \mathbf{c}_{A,A}(a) \ddot{\jmath} a : A}$$

[VCoer Comp / $\text{Quest}_{\mathbf{c}}$]

$$\frac{E \int a:A \quad E \int A <: B \quad E \int B <: C}{E \int \mathbf{c}_{B,C}(\mathbf{c}_{A,B}(a)) \ddot{\jmath} \mathbf{c}_{A,C}(a) : C}$$

The important intuition about coercions is that they involve little, if any, computational work. Often they are introduced as identity functions with the only purpose of "getting the types right". In compilation practice they are often removed during code generation. Semantically, this will be understood in the model for $\text{Quest}_{\mathbf{c}}$ below by observing that they are computed by (indexes of) the identity function. In Quest, the subsumption rule above is a strong (or explicit) way of saying that coercions have no computational relevance.

2.6 Power kinds

For each type A there is a kind P (A) of all subtypes of A. The kind P (Top) is then the kind of all types, and is called T. Here are the formation and introduction rules for P ; the subsumption/coercion rule serves as an elimination rule for P .

[KF P]
$$\frac{E \int A \text{ type}}{E \int P(A) \text{ kind}}$$

[TIncl Refl']
$$\frac{E \int A \text{ type}}{E \int A :: P(A)}$$

The subtype judgment $E \int A <: B$ is defined as an abbreviation for a judgment involving power kinds:

$$E \int A <: B \quad \text{iff} \quad E \int A :: P(B)$$

The subkind judgment $E \int K <:: L$ is primitive, but has very weak properties. It is reflexive and transitive, it extends monotonically to P, and it extends to $\mathbf{P}$ via a covariant rule:

[KIncl P]
$$\frac{E \int A <: A'}{E \int P(A) <:: P(A')}$$

[KIncl $\mathbf{P}$]
$$\frac{E \int K \text{ kind} \quad E, X::K \int L <:: L'}{E \int \mathbf{P}(X::K)L <:: \mathbf{P}(X::K)L'}$$

Note that the first rule above implies $P(A) <:: T$.

Moreover, we have a subsumption rule on kinds:

[KSub] (Kind Subsumption)
$$\frac{E \int A::K \quad E \int K <:: L}{E \int A :: L}$$

Unlike type subsumption, kind subsumption is satisfied by both models in parts 4 and 5.

2.7 Operator kinds

The kind of type operators is normally written as $K \Rightarrow L$ in Fω (operators from kind K to kind L). In our system, as in the Theory of Constructions, we use a more general construction $\mathbf{P}(X::K)L$ since X may actually occur in L within a power operator, for example in $\mathbf{P}(X::T)\,P\,(X)$.

Individual operators are written $\mathbf{l}(X::K)A$ with standard introduction, elimination, and computation rules, shown later.

2.8 The kind of types

The kind of all types T contains the type Top, the types of polymorphic functions, the types of ordinary functions, and the recursive types.

The type Top is the maximal element in the subtype order:

[TF Top] $\dfrac{\int E\ env}{E \int Top\ type}$ [TIncl Top] $\dfrac{E \int A\ type}{E \int A <: Top}$

Hence the power of Top is the collection of all types and, as already mentioned, we can define the kind of all types as follows:

$$T = P(Top)$$

There is a canonical element of type Top, called top. Moreover, any value belonging to Top is indistinguishable from top:

[VI Top] $\dfrac{\int E\ env}{E \int top : Top}$ [VEqTop'] (Top Collapse) $\dfrac{E \int a:Top \quad E \int b:Top}{E \int a = b : Top}$

When using the subsumption rule, we obtain that every value has type Top, since Top is the largest type. Moreover, every value is equivalent to top when seen as a member of Top, and hence $c_{A,Top}(a) = c_{B,Top}(b)$ for any a:A and b:B. By this, when using the coercion rule, there is a unique coercion $c_{A,Top}(a)$ from A into Top. This rather peculiar situation will be understood in the semantics by the meaning of <: and by the interpretation of Top as the terminal object in the intended category. Top and its properties will play a crucial role in the coding of records.

The types of polymorphic functions are modeled by an impredicative general-product construction, P(X::K)B. Although we do not show it here, from this product we can derive "weak" general sums, which are used in the Quest language for modeling abstract types.

The standard formation, introduction, elimination, and computation rules (shown in section 2.9) are complemented by rules for subtyping and coercion:

[TIncl P] $\dfrac{E \int K'<::K \quad E, X::K' \int B<:B'}{E \int P(X::K)B <: P(X::K')B'}$

[VCoer P] $\dfrac{E \int b : P(X::K)B \quad E \int A : K' \quad E \int P(X::K)B <: P(X::K')B'}{E \int (c_{P(X::K)B,P(X::K')B'}(b))(A) = c_{B\{XA\},B'\{XA\}}(b(A)) : B'\{XA\}}$

Ordinary higher-type functions are modeled by a function space construction (). We avoid first-order dependent types (Π(x:A)B, which generalize AB) because in practice they are hard to typecheck and compile. Again, most rules are standard, but we may want to notice subtyping and coercion:

[TIncl]
$$\frac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash A \to B <: A' \to B'}$$

[VCoer]
$$\frac{E \vdash b : A \to B \quad E \vdash a : A' \quad E \vdash A \to B <: A' \to B'}{E \vdash (c_{A \to B, A' \to B'}(b))(a) \equiv c_{B,B'}(b(c_{A',A}(a))) : B'}$$

2.9 Formal system

In this section we summarize the formal systems for both Quest and $Quest_C$. The rules of these systems are presented simultaneously as they largely coincide.

Rules are named, for example, [TExt / Quest] (Type Extension) extra. Here TExt is the proper name of the rule. The notation / Quest means that this rule applies only to Quest, while the notation / $Quest_C$ applies only to $Quest_C$; otherwise the rule applies to both systems. This rule is sometimes called Type Extension in the text. Finally, extra means that this rule is actually derivable or admissible and is listed for symmetry with other rules or for emphasis (for example, [KEq Refl] and [TEq Refl] are provable by simultaneous induction on the derivations).

The rules grouped as "computation" rules may be oriented in order to provide reduction strategies.

A recursive type μ(X)C is legal only if C is *contractive* in X, written CX [MacQueen Plotkin Sethi 86]. A type C is contractive in a (free) type variable X if and only if C has one of the following six forms: a type variable different from X; Top; Π(X'::K)C' with Xfree-variables(K) and C'X; AB; (λ(X'::K)B)(A) with B{X'A}X; or μ(X')C' with C'X (as well as C'X').

We are conservative about the contractiveness conditions on Π(X'::K)C, and these deserve further study. The condition Xfree-variables(K) prevents constructions such as μ(X)Π(Y<:X)XX, whose semantics is unclear. The condition C'X agrees with one of the semantics we give to Π as a non-expansive intersection, although syntactically this restriction seems unnecessary.

Judgments

$\int$ E env	E is an environment
E $\int$ K kind	K is a kind (in an environment E)
E $\int$ A::K	type A has kind K
E $\int$ A type	A is a type (abbr. for E $\int$ A::T)
E $\int$ a:A	value a has type A
E $\int$ K<::L	kind K is a subkind of kind L
E $\int$ A<:B	type A is a subtype of type B (abbr. for E $\int$ A::P (B))
E $\int$ K<::>L	K and L are equivalent kinds
E $\int$ A<:>B::K	A and B are equivalent types or operators of kind K
E $\int$ A<:>B type	A and B are equivalent types (abbr. for E $\int$ A<:>B::T)
E $\int$ a=b:A	a and b are equivalent values

Environments

[Env ∅] $\dfrac{}{\int\ \varnothing \text{ env}}$

[Env X] $\dfrac{E \int K \text{ kind} \quad X \notin \text{dom}(E)}{\int E,X::K \text{ env}}$

[Env x] $\dfrac{E \int A \text{ type} \quad x \notin \text{dom}(E)}{\int E,x:A \text{ env}}$

[Var X] $\dfrac{\int E',X::K,E'' \text{ env}}{E',X::K,E'' \int X :: K}$

[Var x] $\dfrac{\int E',x:A,E'' \text{ env}}{E',x:A,E'' \int x : A}$

Kind formation

[KF P] $\dfrac{E \int A \text{ type}}{E \int P\ (A) \text{ kind}}$

[KF P] $\dfrac{E \int K \text{ kind} \quad E, X::K \int L \text{ kind}}{E \int P(X::K)L \text{ kind}}$

Kind equivalence

[KEq Refl] extra $\dfrac{E \int K \text{ kind}}{E \int K <::> K}$

[KEq Symm] $\dfrac{E \int K <::> L}{E \int L <::> K}$

[KEq Trans] $\dfrac{E \int K <::> L \quad E \int L <::> M}{E \int K <::> M}$

[KEq P] $\dfrac{E \int A<:>A' \text{ type}}{E \int P\ (A) <::> P\ (A')}$

[KEq P] $\dfrac{E \int K <::> K' \quad E, X::K \int L <::> L'}{E \int P(X::K)L <::> P(X::K')L'}$

[KExt] (Kind Extension) extra

$$\frac{E \vdash A::K \quad E \vdash K <::> L}{E \vdash A :: L}$$

Kind inclusion

[KIncl Refl]

$$\frac{E \vdash K <::> L}{E \vdash K <:: L}$$

[KIncl Trans]

$$\frac{E \vdash K <:: L \quad E \vdash L <:: M}{E \vdash K <:: M}$$

[KIncl P]

$$\frac{E \vdash A<:A'}{E \vdash P\ (A) <:: P\ (A')}$$

[KIncl Π]

$$\frac{E \vdash K\ \mathrm{kind} \quad E, X::K \vdash L <:: L'}{E \vdash \Pi(X::K)L <:: \Pi(X::K)L'}$$

[KSub] (Kind Subsumption)

$$\frac{E \vdash A::K \quad E \vdash K<::L}{E \vdash A :: L}$$

Type and Operator formation

[TF Top]

$$\frac{\vdash E\ \mathrm{env}}{E \vdash \mathrm{Top}\ \mathrm{type}}$$

[TF λ]

$$\frac{E, X::T \vdash A\ \mathrm{type}}{E \vdash \lambda(X)A\ \mathrm{type}}$$

[TF Π]

$$\frac{E \vdash K\ \mathrm{kind} \quad E, X::K \vdash B\ \mathrm{type}}{E \vdash \Pi(X::K)B\ \mathrm{type}}$$

[TF _]

$$\frac{E \vdash A\ \mathrm{type} \quad E \vdash B\ \mathrm{type}}{E \vdash AB\ \mathrm{type}}$$

[TI Π]

$$\frac{E \vdash K\ \mathrm{kind} \quad E, X::K \vdash B::L}{E \vdash \lambda(X::K)B :: \Pi(X::K)L}$$

[TE Π]

$$\frac{E \vdash B::\Pi(X::K)L \quad E \vdash A::K}{E \vdash B(A) :: L\{XA\}}$$

Type and Operator equivalence

[TEq Refl] extra

$$\frac{E \vdash A :: K}{E \vdash A <:> A :: K}$$

[TEq Symm]

$$\frac{E \vdash A <:> B :: K}{E \vdash B <:> A :: K}$$

[TEq Trans]

$$\frac{E \vdash A <:> B :: K \quad E \vdash B <:> C :: K}{E \vdash A <:> C :: K}$$

[TEq X]

$$\frac{E \vdash X :: K}{E \vdash X <:> X :: K}$$

[TEq Top]

$$\frac{\vdash E\ \mathrm{env}}{E \vdash \mathrm{Top} <:> \mathrm{Top}\ \mathrm{type}}$$

[TEq Π]

$$\frac{E \vdash K<::>K' \quad E, X::K \vdash B<:>B'\ \mathrm{type}}{E \vdash \Pi(X::K)B <:> \Pi(X::K')B'\ \mathrm{type}}$$

[TEq _]

$$\frac{E \vdash A<:>A'\ \mathrm{type} \quad E \vdash B<:>B'\ \mathrm{type}}{E \vdash AB <:> A'B'\ \mathrm{type}}$$

[TEq Abs]
$$\frac{E \int K <::> K' \quad E, X::K \int B <:> B' :: L}{E \int \lambda(X::K)B <:> \lambda(X::K')B' :: \Pi(X::K)L}$$

[TEq Appl]
$$\frac{E \int B <:> B' :: \Pi(X::K)L \quad E \int A <:> A' :: K}{E \int B(A) <:> B'(A') :: L\{XA\}}$$

[TEq μ]
$$\frac{E, X::T \int B <:> B' \text{ type} \quad B,B'X}{E \int \mu(X)B <:> \mu(X)B' \text{ type}}$$

[TEq Contract]
$$\frac{E \int A <:> C\{XA\} \text{ type} \quad E \int B <:> C\{XB\} \text{ type} \quad CX}{E \int A <:> B \text{ type}}$$

[TExt / Quest] (Type Extension) extra
$$\frac{E \int a:A \quad E \int A <:> B \text{ type}}{E \int a : B}$$

[TExt / Quest$_c$] (Type Extension)
$$\frac{E \int a:A \quad E \int A <:> B \text{ type}}{E \int a : B}$$

[T Π η]
$$\frac{E \int B :: \Pi(X::K)L \quad X \notin dom(E)}{E \int (\lambda(X::K)B(X)) <:> B :: \Pi(X::K)L}$$

Type and Operator computation

[T Π β]
$$\frac{E \int (\lambda(X::K)B)(A) :: L}{E \int (\lambda(X::K)B)(A) <:> B\{XA\} :: L}$$

[T μ]
$$\frac{E, X::T \int A \text{ type} \quad AX}{E \int \mu(X)A <:> A\{X\mu(X)A\} \text{ type}}$$

Type inclusion

[TIncl Refl]
$$\frac{E \int A <:> B \text{ type}}{E \int A <: B}$$

[TIncl Trans]
$$\frac{E \int A <: B \quad E \int B <: C}{E \int A <: C}$$

[TIncl Top]
$$\frac{E \int A \text{ type}}{E \int A <: \text{Top}}$$

[TIncl Π]
$$\frac{E \int K'<::K \quad E, X::K' \int B<:B'}{E \int \Pi(X::K)B <: \Pi(X::K')B'}$$

[TIncl]
$$\frac{E \int A'<:A \quad E \int B<:B'}{E \int AB <: A'B'}$$

[TIncl μ]
$$\frac{E \int \mu(X)A \text{ type} \quad E \int \mu(Y)B \text{ type} \quad E, Y::T, X<:Y \int A <: B}{E \int \mu(X)A <: \mu(Y)B}$$

[TSub / Quest] (Subsumption)
$$\frac{E \int a:A \quad E \int A<:B}{E \int a : B}$$

[TSub / Quest$_c$] (Coercion)
$$\frac{E \int a:A \quad E \int A<:B}{E \int c_{A,B}(a) : B}$$

Value formation

[VI Top]

$$\frac{\int E\ \text{env}}{E \int \text{top} : \text{Top}}$$

[VI Π]

$$\frac{E \int K\ \text{kind} \quad E, X::K \int b:B}{E \int \lambda(X::K)b : \Pi(X::K)B}$$

[VE Π]

$$\frac{E \int b:\Pi(X::K)B \quad E \int A::K}{E \int b(A) : B\{XA\}}$$

[VI $\rightarrow$]

$$\frac{E \int A\ \text{type} \quad E, x:A \int b:B}{E \int \lambda(x:A)b : A \rightarrow B}$$

[VE $\rightarrow$]

$$\frac{E \int b:A \rightarrow B \quad E \int a:A}{E \int b(a) : B}$$

[VI c / Quest$_c$]

$$\frac{E \int A <: B \quad E \int a:A}{E \int c_{A,B}(a) : B}$$

[VI μ]

$$\frac{E \int A\ \text{type} \quad E, x:A \int b:A}{E \int \mu(x:A)b : A}$$

Value equivalence

[VEq Refl] extra

$$\frac{E \int a : A}{E \int a \equiv a : A}$$

[VEq Symm]

$$\frac{E \int a \equiv b : A}{E \int b \equiv a : A}$$

[VEq Trans]

$$\frac{E \int a \equiv b : A \quad E \int b \equiv c : A}{E \int a \equiv c : A}$$

[VEqSub / Quest] (Subsumption Eq)

$$\frac{E \int a \equiv a' : A \quad E \int A <: B}{E \int a \equiv a' : B}$$

[VEqSub / Quest$_c$] (Coercion Eq)

$$\frac{E \int a \equiv a' : A \quad E \int A <: B \quad E \int A <:> A'\ \text{type} \quad E \int B <:> B'\ \text{type}}{E \int c_{A,B}(a) \equiv c_{A',B'}(a') : B}$$

[VEq x]

$$\frac{E \int x:A}{E \int x \equiv x : A}$$

[VEq top]

$$\frac{\int E\ \text{env}}{E \int \text{top} \equiv \text{top} : \text{Top}}$$

[VEqTop] (Top Collapse)

$$\frac{E \int a \equiv a : \text{Top} \quad E \int b \equiv b : \text{Top}}{E \int a \equiv b : \text{Top}}$$

[VEq TAbs]

$$\frac{E \int K <::> K' \quad E, X::K \int b \equiv b' : B}{E \int \lambda(X::K)b \equiv \lambda(X::K')b' : \Pi(X::K)B}$$

[VEq TAppl]

$$\frac{E \int b \equiv b' :: \Pi(X::K)B \quad E \int A <:> A' :: K}{E \int b(A) \equiv b'(A') : B\{XA\}}$$

[VEq Abs]

$$\frac{E \int A <:> A'\ \text{type} \quad E, x:A \int b \equiv b' : B}{E \int \lambda(x:A)b \equiv \lambda(x:A')b' : A \rightarrow B}$$

[VEq Appl]

$$\frac{E \int b \equiv b' : A \rightarrow B \quad E \int a \equiv a' : A}{E \int b(a) \equiv b'(a') : B}$$

[VEq $\mathbf{m}$]

$$\frac{E \int A <:> A' \text{ type} \quad E, x{:}A \int b \ddot{\mathbf{y}} b' : A}{E \int \mathbf{m}(x{:}A)b \ddot{\mathbf{y}} \mathbf{m}(x{:}A')b' : A}$$

[$\mathbf{P}$ $\mathbf{h}$ / Quest$_{\mathbf{c}}$]

$$\frac{E \int b : \mathbf{P}(X{::}K)B \quad X \notin dom(E)}{E \int (\mathbf{l}(X{::}K)b(X)) \ddot{\mathbf{y}} b : \mathbf{P}(X{::}K)B}$$

[$\mathbf{h}$ / Quest$_{\mathbf{c}}$]

$$\frac{E \int b : AB \quad x \notin dom(E)}{E \int (\mathbf{l}(x{:}A)b(x)) \ddot{\mathbf{y}} b : AB}$$

Value coercion

[VCoer Id / Quest$_{\mathbf{c}}$]

$$\frac{E \int a{:}A}{E \int \mathbf{c}_{A,A}(a) \ddot{\mathbf{y}} a : A}$$

[VCoer Comp / Quest$_{\mathbf{c}}$]

$$\frac{E \int a{:}A \quad E \int A <: B \quad E \int B <: C}{E \int \mathbf{c}_{B,C}(\mathbf{c}_{A,B}(a)) \ddot{\mathbf{y}} \mathbf{c}_{A,C}(a) : C}$$

[VCoer Top / Quest$_{\mathbf{c}}$] extra

$$\frac{E \int a : A}{E \int \mathbf{c}_{A,Top}(a) \ddot{\mathbf{y}} top : Top}$$

[VCoer $\mathbf{P}$ / Quest$_{\mathbf{c}}$]

$$\frac{E \int b : \mathbf{P}(X{::}K)B \quad E \int A : K' \quad E \int \mathbf{P}(X{::}K)B <: \mathbf{P}(X{::}K')B'}{E \int (\mathbf{c}_{\mathbf{P}(X::K)B,\mathbf{P}(X::K')B'}(b))(A) \ddot{\mathbf{y}} \mathbf{c}_{B\{XA\},B'\{XA\}}(b(A)) : B'\{XA\}}$$

[VCoer / Quest$_{\mathbf{c}}$]

$$\frac{E \int b : AB \quad E \int a : A' \quad E \int AB <: A'B'}{E \int (\mathbf{c}_{AB,A'B'}(b))(a) \ddot{\mathbf{y}} \mathbf{c}_{B,B'}(b(\mathbf{c}_{A',A}(a)) : B'}$$

[VCoer $\mathbf{m}$ / Quest$_{\mathbf{c}}$]

$$\frac{E \int a : \mathbf{m}(X)A \quad E \int A : K' \quad E \int \mathbf{m}(X)A <: \mathbf{m}(Y)B}{E \int \mathbf{c}_{\mathbf{m}(X)A,\mathbf{m}(Y)B}(a) \ddot{\mathbf{y}} \mathbf{c}_{A\{X\mathbf{m}(X)A\},B\{Y\mathbf{m}(Y)\}}(a) : \mathbf{m}(Y)B}$$

Value computation

[$\mathbf{P}$ $\mathbf{b}$]

$$\frac{E \int (\mathbf{l}(X{::}K)b)(A) : B}{E \int (\mathbf{l}(X{::}K)b)(A) \ddot{\mathbf{y}} b\{XA\} : B}$$

[$\mathbf{b}$]

$$\frac{E \int (\mathbf{l}(x{:}A)b)(a) : B}{E \int (\mathbf{l}(x{:}A)b)(a) \ddot{\mathbf{y}} b\{xa\} : B}$$

[$\mathbf{m}$]

$$\frac{E \int \mathbf{m}(x{:}A)b : A}{E \int \mathbf{m}(x{:}A)b \ddot{\mathbf{y}} b\{x\mathbf{m}(x{:}A)b\} : A}$$

2.10 Records and other encodings

Record types are one of the main motivations for studying type systems with subtyping [Cardelli 88]. However, in this paper we do not need to model them directly (as already done in [Bruce Longo 89]), since they can be syntactically encoded to a great extent.

More precisely, we show how to encode the record calculus of [Cardelli Wegner 85], although we do not know yet how to encode the more powerful calculi of [Wand 89] and [Cardelli Mitchell 89]. Moreover, we show how to encode the *functional update* problem discussed in [Cardelli Mitchell 89]; this problem cannot be represented in the calculus of [Cardelli Wegner 85].

In this section we discuss these encodings, and then we feel free to ignore records in the rest of the paper.

We start by encoding product types, in the usual way:

$$A\ B \qquad \forall(C)(A \to B \to C) \to C$$

$$\begin{aligned}
\text{pair} &: \forall(A)\ \forall(B)\ A \to B \to A\ B \\
&\quad \lambda(A)\ \lambda(B)\ \lambda(a{:}A)\ \lambda(b{:}B)\ \lambda(C)\ \lambda(f{:}A \to B \to C)\ f(a)(b) \\
\text{fst} &: \forall(A)\ \forall(B)\ A\ B \to A \\
&\quad \lambda(A)\ \lambda(B)\ \lambda(c{:}A\ B)\ c(A)(\lambda(x{:}A)\lambda(y{:}B)x) \\
\text{snd} &: \forall(A)\ \forall(B)\ A\ B \to B \\
&\quad \lambda(A)\ \lambda(B)\ \lambda(c{:}A\ B)\ c(B)(\lambda(x{:}A)\lambda(y{:}B)y)
\end{aligned}$$

We often use a more compact notation:

a,b	$a,_{AB}b$	pair(A)(B)(a)(b)
fst(c)	$fst_{AB}(c)$	fst(A)(B)(c)
snd(c)	$snd_{AB}(c)$	snd(A)(B)(c)

The expected rules for products are now derivable:

$$\frac{E \vdash A <: A' \quad E \vdash B <: B'}{E \vdash A\ B <: A'\ B'}$$

$$\frac{E \vdash P <: A\ B \quad E \vdash p{:}P}{E \vdash fst_{AB}(p) : A} \qquad \frac{E \vdash P <: A\ B \quad E \vdash p{:}P}{E \vdash snd_{AB}(p) : B}$$

As a first step toward records, we define *extensible tuple types* as iterated products ending with Top, and *extensible tuple values* as iterated pairs ending with top. A similar encoding appears in [Fairbairn 89].

$$Tuple(A_1,...,A_n) \qquad A_1(...(A_n\ Top)..)$$

$\text{tuple}(a_1,...,a_n)$ $\quad$ $a_1,(...,(a_n, \text{top})..)$

Hence:

$$\frac{E \int a_1 : A_1 \; ... \; E \int a_n : A_n}{E \int \text{tuple}(a_1,...,a_n) : \text{Tuple}(A_1,...,A_n)}$$

$$\frac{E \int A_1 <: B_1 \; ... \; E \int A_n <: B_n \; ... \; E \int A_m \text{ type}}{E \int \text{Tuple}(A_1,...,A_n,...,A_m) <: \text{Tuple}(B_1,...,B_n)}$$

For example: Tuple(A, B) <: Tuple(A) since A <: A, BTop <: Top, and is monotonic.

We now need to define tuple *selectors* (corresponding to product projections). This would be a family sel_i^n of terms selecting the i-th components of a tuple of length n. In fact, by using subtyping it is sufficient to define a family sel_i of terms for extracting the i-th component of any tuple of sufficient length:

sel_1 : $\quad \forall(A_1)\ A_1 \text{Top} A_1$
$\quad\quad\quad \lambda(A_1)\ \lambda(t{:}A_1\text{Top})\ \text{fst}_{A_1\text{Top}}(t)$

sel_2 : $\quad \forall(A_2)\ \text{Top}A_2\text{Top}A_2$
$\quad\quad\quad \lambda(A_2)\ \lambda(t{:}\text{Top}A_2\text{Top})$
$\quad\quad\quad\quad \text{fst}_{A_2\text{Top}}(\text{snd}_{\text{Top}A_2\text{Top}}(t))$
etc.

We can also define tuple *updators*, that is, terms that replace the i-th component of a tuple with a given value. The crucial point here is that these updators do not forget information about the type of the components that are not affected by the update. To achieve this effect, we must use knowledge of the encoding of tuples as pairs. Again, we can define a family upd_i instead of a family upd_i^n.

upd_1 : $\quad \forall(B_1)\forall(B_{tl})\forall(A_1)\ B_1 B_{tl} A_1 A_1 B_{tl}$
$\quad\quad\quad \lambda(B_1)\lambda(B_{tl})\lambda(A_1)$
$\quad\quad\quad\quad \lambda(t{:}B_1B_{tl})\lambda(a_1{:}A_1)\ a_{1,A_1B_{tl}}\ \text{snd}_{B_1B_{tl}}(t)$

upd_2 : $\forall(B_1)\,\forall(B_2)\,\forall(B_{tl})\,\forall(A_2)\;\; B_1B_2B_{tl}A_2B_1A_2B_{tl}$
$\lambda(B_1)\,\lambda(B_2)\,\lambda(B_{tl})\,\lambda(A_2)$
$\lambda(t{:}B_1B_2B_{tl})\,\lambda(a_2{:}A_2)$ fst(t),(a_2, snd(snd(t)))
etc.

These definitions solve the functional update problem [Cardelli Mitchell 89] for tuples. This problem can be explained by the following example, where we update a field of a tuple in such a way that the updating function works equally well on subtypes of the stated tuple type.

We have a type of geometric points defined as Point = Tuple(Int,Int), where the integers represent respectively the x and y components. Since these are tuples, a point can have additional components, for example a color; then it is a member of ColorPoint = Tuple(Int,Int,Color). We further assume that the subrange type 0..9 is a subtype of Int.

The problem consists in defining a function moveX that increments the x component of a point, returning another Point. Moreover, when applied to a ColorPoint (with adequate type parameters) this function should return a ColorPoint, and not just a Point.

One might think that moveX has type $\forall(A{<:}\text{Point})\;\; AA$. This is not the case; we show that the parameter type A must change appropriately from input to output.

Point Tuple(Int,Int)

moveX : $\forall(B_1{<:}\text{Int})\;\forall(B_{tl}{<:}\text{Tuple(Int)})\;\; B_1B_{tl}\text{Int}B_{tl}$
$\lambda(B_1{<:}\text{Int})\;\lambda(B_{tl}{<:}\text{Tuple(Int)})\;\lambda(p{:}B_1B_{tl})$
$upd_1(B_1)(B_{tl})(\text{Int})(p)(sel_1(\text{Int})(p)+1)$

Obviously, we have:

p : Point tuple(9,0)
moveX(Int)(Tuple(Int))(p) tuple(10,0) : Point

However, note that in the following example the result does not, and must not, have type Tuple(0..9,Int):

p : Tuple(0..9,Int) <: Point tuple(9,0)
moveX(0..9)(Tuple(Int))(p) tuple(10,0) : Point

We can also verify that color is preserved:

p : Tuple(0..9,Int,Color) <: ColorPoint tuple(9,0,red)
moveX(0..9)(Tuple(Int,Color))(p) tuple(10,0,red) : ColorPoint

Hence, we obtain a moveX function with the desired properties, but only by taking advantage of the encoding of tuples as products. Note that in the input type of moveX, Point is split into Int and Tuple(Int).

Now we turn to the encoding of records $Rcd(l_1:A_1, \ldots, l_n:A_n)$; these are unordered product types with components indexed by distinct labels l_i.

We fix a standard enumeration of labels $l^1, l^2, \ldots$. Then a record type is the shortest tuple type where the type component of label l^i is found in the tuple slot of index i, for each i. The remaining slots are filled with Top. For example:

$Rcd(l^3:C, l^1:A)$ Tuple(A, Top, C)

Under this encoding, record types that differ only on the order of components are equivalent, and we have the familiar:

$$\frac{E \int A_1 <: B_1 \ \ldots \ E \int A_n <: B_n \ \ldots \ E \int A_m \text{ type}}{E \int Rcd(l_1:A_1,\ldots,l_n:A_n,\ldots,l_m:A_m) <: Rcd(l_1:B_1,\ldots,l_n:B_n)}$$

Record values are similarly encoded, for example:

$rcd(l^3=c, l^1=a)$ tuple(a, top, c)

$$\frac{E \int a_1 : A_1 \ \ldots \ E \int a_n : A_n}{E \int rcd(l_1=a_1,\ldots,l_n=a_n) : Rcd(l_1:A_1,\ldots,l_n:A_n)}$$

$$\frac{E \int r : Rcd(l_1:A_1,\ldots,l_1:A_n)}{E \int r.l_i : A_i} \qquad \frac{E \int r : Rcd(l_1:A_1,\ldots,l_i:A_i,\ldots,l_1:A_n) \quad E \int b:B}{E \int r.l_i b : Rcd(l_1:A_1,\ldots,l_i:B,\ldots,l_n:A_n)}$$

Here record selection $r.l_i$ is defined via $sel_i(r)$, and record update $r.l_i b$ is defined via $upd_i(r)(b)$.

Note that it is not possible to write a version of moveX for records solely by using the derived operators above. The functional update problem can be solved only by using knowledge of the encodings, as was done for tuples. In

this respect (an encoding of) a calculus like the one in [Cardelli Mitchell 89] is still to be preferred, since it can express the moveX functions independently of encodings.

Under the encodings above, more programs are typable than we would normally desire; this is to be expected of any encoding strategy. The important point here is that the familiar typing and computation rules are sound.

3. PER and ω-Set

The rest of the paper describes the mathematical meaning of the Quest system described in the previous section. The goal here is to guarantee the (relative) consistency of Quest's type and equational theories. The model though is also meant to suggest consistent extensions. This is one of the reasons for which we construct a specific (class of) model(s), instead of suggesting general definitions. These may be obtained by slight modifications of the work in [Bruce Longo 88] , or, even better, by following the categorical approach in [Asperti Longo 91]. Indeed, in the latter case, the invention of a general categorical meaning for subtyping and subkinds would be a relevant contribution.

In this part, we first try to give the structural (and partly informal) meaning of kinds, types, and terms, as well as their crucial properties. The reader will find the properties formally described in part 2 reflected over sets and functions, and should grasp the essence of the translation. Part 4 develops further the details of the interpretation of $Quest_C$ that the experienced reader could give by himself, at that point. Part 5 describes Quest with the subsumption rule, instead of with coercions.

Because of the presence of type operators, the structure of kinds is at least as rich as the type-structure of typed λ-calculus. Thus, kinds need to be interpreted as objects of a Cartesian Closed Category, CCC. The category we will be using is **ω-Set** below. Its objects must, of course, include the kind of types, which in turn must be structured as a CCC.

In a sense, we need a *frame* (or *global*) *category* , inside which we may view the category of types as an object. More precisely, we need a frame category and an *internal category*, but we will not go into this here, except in remark 3.1.5. The general approach by internal categories was suggested by Moggi and has been developed by several authors; see 3.1.5 for references.

The specific structures used here, that is **ω-Set** and **PER** below, are described in [Longo Moggi 88], where their main categorical properties are also

given. The approach in [Longo Moggi 88] is elementary: indeed, these categories may be seen as subcategories of Hyland's Effective Topos (see [Hyland 82 and 87] for the topos theoretic approach). The idea of interpreting subtypes as subrelations is borrowed from [Bruce Longo 89], where the semantics of Quest's progenitor system, Bounded Fun (with coercions), was first given.

3.1 Semantics of kinds and types

The key idea in the underlying mathematical construction is to use a set-theoretic approach where the addition of some *effectiveness* prevents the difficulties discussed in [Reynolds 84]. In this regard, the blend of set-theoretic intuition and elementary computability provides a simple but robust guideline for the interpretation of programming constructs.

The construction is based on Kleene's applicative structure $(\omega, \cdot)$, where ω is the set of natural numbers, together with a standard gödelization φ_n of the computable functions in $\omega\omega$, and where $\cdot$ is the operator such that $n \cdot m = \varphi_n(m)$. However, the same mathematical construction works for any (possibly partial) combinatory algebra, in particular on any model of type-free λ-calculus. We prefer, in this part, Kleene's $(\omega, \cdot)$ in view of everybody's familiarity with elementary recursion theory. In part 5, though, we will base our construction on models of the type-free λ-calculus.

Definition 3.1.1

The category **ω-Set** has:

objects: $\langle A, \Vdash_A \rangle \in$ **ω-Set** iff

A is a set and $\Vdash_A \subseteq \omega \times A$ is a relation, such that $\forall a \in A.\ \exists n.\ n \Vdash_A a$

morphisms: $f \in$ **ω-Set**$[A,B]$ iff

$f: A \to B$ and $\exists n.\ n \Vdash_{A \to B} f$,

where $n \Vdash_{A \to B} f \ \Leftrightarrow\ \forall a \in A.\ \forall p.\ p \Vdash_A a \ \Rightarrow\ n \cdot p \Vdash_B f(a)$

Thus, each morphism in **ω-Set** is "computable" in the sense that it is described by a partial recursive function that is total on $\{p \mid p \Vdash_A a\}$, for each $a \in A$. If $p \Vdash a$ (we may omit the subscripts), we say that p *realizes* a (or p *computes* a).

We next define the category of types. When A is a symmetric and transitive relation on ω, we set:

n A m iff n is related to m by A,
dom(A) $= \{n \mid n\ A\ n\}$,
$\ulcorner \mathbf{n} \urcorner_{\mathbf{A}} = \{m \mid m\ A\ n\}$ the equivalence class of n with respect to A,
Q(A) $= \{\ulcorner n \urcorner_A \mid n \in \mathrm{dom}(A)\}$ the quotient set of A.

Definition 3.1.2

The category **PER** (of Partial Equivalence Relations) has
objects: A∈**PER** iff

A is a symmetric and transitive relation on ω,
morphisms: f∈**PER**[A,B] iff

f: Q(A) Q(B) and $\exists n.\ \forall p.\ (pAp \Rightarrow f(\{p\}_A) = \{n \cdot p\}_B)$

PER is a category where the identity map, in each type, is computed by (at least) any index of the identity function on ω.

The category **PER** can be fully and faithfully embedded into **ω-Set**. In fact, for every partial equivalence relation (p.e.r.) A, define the ω-set In(A) = $\langle Q(A), \in_A \rangle$, where Q(A) are the equivalence classes of A as subsets of ω, and $\in_A$ is the usual membership relation restricted to $\omega \times Q(A)$. Clearly, $\in_A$ defines a realizability relation in the sense of 3.1.1 and the functor In is full and faithful. Note that $\in_A$ is a single-valued relation, as equivalence classes are disjoint subsets of ω.

The following simple fact may help in identifying which are the maps in **PER**, by viewing them also as morphisms in **ω-Set**. (The reader should practice going from one category to the other; the next proposition is just an exercise with this purpose.)

Proposition 3.1.3

Let f ∈ **PER**[A,C], then

$p \Vdash_{AC} f$ (in **ω-Set**) $\Leftrightarrow$ $\forall r.\ (r\ A\ r \Rightarrow \{p \cdot r\}_C = f(\{r\}_A))$

Proof

$p \Vdash_{AC} f \Leftrightarrow \forall a \in Q(A).\ \forall r \Vdash_A a.\ p \cdot r \Vdash_C f(a)$

$\Leftrightarrow \forall r.\ r\ A\ r \Rightarrow p \cdot r \in f(\{r\}_A)$,

since $\Vdash$ coincides with ∈ (with respect to an equivalence class).
Hence we must show:

$\forall r.\ (r\ A\ r \Rightarrow p \cdot r \in f(\{r\}_A)) \Leftrightarrow \forall r.\ (r\ A\ r \Rightarrow \{p \cdot r\}_C = f(\{r\}_A))$.

Case ⇐) Obvious, since $p \cdot r \in \{p \cdot r\}_C$

Case ⇒) Suppose $\{p \cdot r\}_C \neq f(\{r\}_A)$, then $\{p \cdot r\}_C \cap f(\{r\}_A) = \emptyset$ since Q(C) is a quotient, but $p \cdot r \in \{p \cdot r\}_C$, and by hypothesis $p \cdot r \in f(\{r\}_A)$. Contradiction.

What is relevant for us, though, is that **PER** may be viewed also as an object of **ω-Set**; this interprets the fact that T is a kind. The point is that the objects of **PER** form a set and every set may be viewed as an ω-set:

Definition 3.1.4

Let $\mathbb{D}$: **Set** $\to$ **ω-Set** be given by $\mathbb{D}(S) = \langle S, \mathcal{E}_S \rangle$, where $\mathcal{E}_S = \omega \times S$, that is, $\forall n\ \forall s$ $n \mathcal{E}_S s$ (the full relation). The function $\mathbb{D}$ is extended to a functor by setting $\mathbb{D}(f) = f$, the identity on morphism.

In particular, set $\mathbf{M_o} = \mathbb{D}(\mathbf{PER}) \in$ **ω-Set,** the ω-set of types.

Remark 3.1.5 (For readers with some experience in Category Theory.)

ω-Set was equivalently defined in [Hyland 82] as the "~~separated objects" in his Effective Topos, **Eff**. The category **ω-Set** has all finite limits and is a locally CCC (see below for the cartesian closure). The embedding $\mathbb{D}$ above preserves exponents and limits. Moreover, one may embed **ω-Set** into **Eff** by a functor which preserves limits and the lCCC structure.

By this, the present approach applies in a simple set-theoretic framework the results in [Hyland 87], [Pitts 87], [Hyland Pitts 87], [Carboni Freyd Scedrov 87], and [Bainbridge Freyd Scedrov Scott 87]. The general treatment of models, as internal categories of categories with finite limits, which was suggested by Moggi, is given in [Asperti Martini 89] and [Asperti Longo 90]. The elegant presentation in [Meseguer 88] compares various approaches. We use here the fact that **ω-Set** is closed under products *indexed over itself* and, in particular, we use the completeness of **PER** as an internal category. The categorical products are exactly those naively defined below (to within isomorphism). Both the explicit definition of **PER** as an internal category and the required (internal) adjunctions are given in detail in [Longo Moggi 88], which is written also for non category-theorists. (See also [Asperti Longo 90].)

The reason for the next definitions is that we need to be able to give meaning, over these structures, to kinds and types constructed as products, as expressed in rules [KF $\mathbb{P}$] and [TF $\mathbb{P}$] in section 2.9. We take care of this point first, since it deals with the crucial aspect of impredicativity in Quest. A first idea is to try to understand those rather complex kinds and types as indexed products, in the naive sense of set theory. Namely, given a set A and a function G: A $\to$ **Set,** define as usual:

$$\times_{a \in A} G(a) = \{f \mid f: A \to \bigcup_{a \in A} G(a) \text{ and } f(a) \in G(a)\}.$$

This product wouldn't work, but the following simple restriction to realizable maps f, will work.

Definition 3.1.6

Let $\langle A, \Vdash_A \rangle \in \omega\textbf{-Set}$ and $G: A \to \omega\textbf{-Set}$. Define the ω-set $\langle \prod_{a\in A} G(a), \Vdash_{\prod G} \rangle$ by

1) $f \in \prod_{a\in A} G(a)$ iff $f \in \times_{a\in A} G(a)$ and $\exists n.\ \forall a\in A.\ \forall p \Vdash_A a.\ n \cdot p \Vdash_{G(a)} f(a)$,
2) $n \Vdash_{\prod G} f$ iff $\forall a\in A.\ \forall p \Vdash_A a.\ n \cdot p \Vdash_{G(a)} f(a)$

When the range of G is restricted to **PER** we obtain a product in **PER**:

Definition 3.1.7

Let $\langle A, \Vdash_A \rangle \in \omega\textbf{-Set}$ and $G: A \to \textbf{PER}$. Let $\prod_{a\in A} G(a)_{\textbf{PER}} \in \textbf{PER}$ be defined by

$n\ (\prod_{a\in A} G(a)_{\textbf{PER}})\ m$ iff $\forall a\in A.\ \forall p,q \Vdash_A a.\ n \cdot p\ G(a)\ m \cdot q$

A crucial property of ω**-Set** is that the products defined in 3.1.6 and 3.1.7 are isomorphic for $G: A \to \textbf{PER}$.

Theorem 3.1.8 ([Bruce Longo 89])

Let $\langle A, \Vdash_A \rangle \in \omega\textbf{-Set}$ and $G: A \to \textbf{PER}$. Then

$\langle \prod_{a\in A} \mathrm{In}(G(a)), \Vdash_{\prod G} \rangle \cong \mathrm{In}(\prod_{a\in A} G(a)_{\textbf{PER}})$ in $\omega\textbf{-Set}$.

Proof

Let $\Vdash_{\prod G}$ be defined as in 3.1.6. We first prove that $\Vdash_{\prod G}$ is a single-valued relation. Assume that $n \Vdash_{\prod G} f \wedge n \Vdash_{\prod G} h$. We show that $\forall a\in A.\ f(a) = h(a)$ and thus, that $f = h$. By definition $\forall a\in A.\ \forall p \Vdash_A a.\ n \cdot p \Vdash_{G(a)} f(a) \wedge n \cdot p \Vdash_{G(a)} h(a)$, and thus $f(a) = h(a)$ since, for all a, the relation $\Vdash_{G(a)}$ is single valued (and any a in A is realized by some natural number).

The isomorphism is given by $J(f) = \{n \mid n \Vdash_{\prod G} f\}$; thus the range of J is a collection of disjoint sets in ω (equivalence classes). The isomorphism J and its inverse are realized by the (indices for the) identity function.

The existence in **PER** of "products" indexed over arbitrary ω-sets is a very relevant fact. The point is to show that these object are real products, in a precise categorical sense; this is hinted in remark 3.15. What we can do here, in our elementary approach, is to use the idea in definition 3.1.7, in order to construct exponents as particular cases of products.

Corollary 3.1.9

ω**-Set** and **PER** are CCC's. Moreover, the embedding In: **PER** $\to$ ω**-Set** is full, faithful and preserves the structure of CCC.

Proof

Observe that if $G: A \to \omega\textbf{-Set}$ is a constant function, $G(a) = \langle B, \Vdash_B \rangle$ for all $a\in A$, say, then $\langle \prod_{a\in A} G(a), \Vdash_{\prod G} \rangle = \langle B^A, \Vdash_{AB} \rangle$ is the exponent representing ω-

Set[A,B] in **ω-Set**. Clearly, in that case, n $\Vdash_{AB}$ f iff ∀a∈A. ∀p $\Vdash_A$ a. n•p $\Vdash_B$ f(a). Products are defined by using any bijective pairing functions from ω×ω to ω. Any singleton set S gives a terminal object **D**(S). Eval and the currying operation **Λ** are defined as in **Set** and are realized by (the indexes of) the universal function and the function s of the s-m-n theorem. (The reader may check this as an exercise or see [Asperti Longo 90] for details.)

The same argument applies to **PER** by taking, for A∈**PER**, G: A **PER** constant in 3.1.8. (Just recall that **PER** may be viewed as the ω-set $\mathbf{M_0}$ = **D**(**PER**) and set ⟨A, $\Vdash_A$⟩ = $\mathbf{M_0}$.) Or also, by embedding **PER** in **ω-Set** by In, the corresponding ω-sets give exponents, products, and terminal objects (up to isomorphisms), as In trivially satisfies the properties stated.

To clarify the construction, let's look more closely to exponent objects in **PER**. Take say AB, that is, the representative of **PER**[A,B]. Then by definition each map f∈**PER**[A,B] is uniquely associated with the equivalence class of its realizers, $\ulcorner p \urcorner_{AB} \in AB$, say, in the sense of 3.1.3.

It should be clear that the notion of realizer, or "type-free computation" computing the typed function, is made possible by the underlying type-free universe, (ω,·). As we will discuss later, this gives mathematical meaning to the intended type-free computations of a typed program after compilation. As for now, this feature of the realizability model suggests a distinction between isomorphism in our categories, which does not need to make sense in other frames (and is relevant for the intuition on which our mathematical understanding is based):

Definition 3.1.10

An isomorphism f: A ≅ B in **ω-Set** is *identical* (or is an identical isomorphism) if both f and its inverse f^{-1} are realized by the indices of the identity function.

It is easy to rephrase this notion for objects in **PER**. Note though that A ≅ B in **PER** via an identical isomorphism iff A = B (that is, A and B are equal).

In **ω-Set**, though, the isomorphism in 3.1.8 is identical (but it is not an identity).

Proposition 3.1.11

In: **PER** **ω-Set** preserves products and exponents to within identical isomorphism.

Proof

Exercise. (The category oriented reader may check these preservation

properties also for equalizers, limits... and observe that they are generally not *on the nose*.)

In summary, our types may be essentially viewed as kinds, by a very natural (and strong) embedding. We applied this embedding in theorem 3.1.8, and gave there a unified understanding of various products and arrows in the syntax. However, theorem 3.1.8 really leads to much more than the cartesian closure of **PER,** which is shown in corollary 3.1.9. In plain terms, 3.1.8 is the crucial step towards the meaning of the second-order (polymorphic) types, namely of the types obtained by indexing a collection of types over a kind, possibly over the collection of all types (an impredicative construction).

3.2 Inclusion and power kinds

The purpose of this section is to set the basis for the semantics of the subkind and subtype relations in Quest.

Definition 3.2.1 (subkinds)

Let $\langle A, \Vdash_A\rangle, \langle B, \Vdash_B\rangle \in \omega\text{-}\mathbf{Set}$. Define:

$\langle A, \Vdash_A\rangle \leq \langle B, \Vdash_B\rangle$ iff $A \subseteq B$ and $\forall a \in A.\forall n.\ (n \Vdash_A a \Rightarrow n \Vdash_B a)$

The idea in this definition is that kinds may be related by the $\leq$ relation in ω**-Set** only when they are actually subsets and when the realizability relation is defined in accordance with this. Thus there is no need of coercions (equivalently, coercions are just identity functions). Hence, the subsumption rule [KSub] for kinds is realized. Subtyping will be interpreted in **PER** in a more subtle way, which allows a closer look at the computational properties of the types of programs.

Definition 3.2.2: (subtypes)

Let $A, B \in \mathbf{PER}$. Define:

$A \leq B$ iff $\forall n,m.\ (n\ A\ m \Rightarrow n\ B\ m)$

Both $\leq$ relations in ω**-Set** and **PER** are reflexive and transitive. They are even antisymmetric, because for $\langle A, \Vdash_A\rangle, \langle B, \Vdash_B\rangle \in \omega\text{-}\mathbf{Set}$ we have $\langle A, \Vdash_A\rangle = \langle B, \Vdash_B\rangle$ $\langle A, \Vdash_A\rangle \leq \langle B, \Vdash_B\rangle$ $\langle B, \Vdash_B\rangle \leq \langle A, \Vdash_A\rangle$. Similarly, for $C,D \in \mathbf{PER}$ we have $C = D$ $C \leq D$ $D \leq C$.

The semantic notion of subtype we are using here is the one defined in [Bruce Longo 89]. However, we differ from that approach for subkinds, in order to model the strong relation we formalized in the syntax of Quest.

Clearly "≤" is a partial order which turns the objects of **PER** into an algebraic complete lattice. When A and B are in **PER** and A ≤ B, then there is a *coercer* $\mathbf{c}_{A,B}$ from A to B. It is defined by the map $\mathbf{c}_{A,B}$: Q(A) Q(B) such that $\mathbf{c}_{A,B}(\ulcorner n \urcorner_A) = \ulcorner n \urcorner_B$, which is computed by any index of the identity function. By definition, $\mathbf{c}_{A,B}$ is uniquely determined by A and B. (We may omit the subscripts, if there is no ambiguity.)

Intuitively, given n such that nAn, the coercion $\mathbf{c}_{A,B}$ takes its A-equivalence class, $\ulcorner n \urcorner_A$, to its (possibly larger) B-equivalence class, $\ulcorner n \urcorner_B$. This is why $\mathbf{c}_{A,B}$, the coercion morphism, is computed by all the indices of the identity function. Note that in general $\ulcorner n \urcorner_A$ is smaller than $\ulcorner n \urcorner_B$; they coincide just when Q(A) ⊆ Q(B), a special case of A ≤ B. Note also that for A,B∈**PER**, if In(A) ≤ In(B) regarded as ω-sets, then A ≤ B. The reverse implication holds only when Q(A) ⊆ Q(B). The result is that, here, ≤ is used with a slightly different meaning in the two categories, in contrast to the approach in [Bruce Longo 89]. The advantage is given by the construction of a model of the current rich kind and type theory.

The power operation is expressed in terms of *quasi-functors*, a weak notion of categorical transformation between categories, widely used in several settings. (See [Martini 88] for recent applications to the semantics of the λ-calculus.) This interpretation is due to the blend of set-theoretical and categorical intuition at the base of the current model of subtyping in a higher-order language. Quasi-functors take morphisms to sets of morphisms which behave consistently with respect to application (see below), and are such that the image of each identity map contains the identity in the target category.

Definition 3.2.3

The *power* quasi-functor P : **PER** **ω-Set** is given by:
on objects: P A = ⟨{B∈**PER** | B ≤ A}, ε⟩, where ∀B ≤ A ∀n n ε B;
on morphisms: for f: AC and p ε f, define P_p(f): P AP C pointwise by
mP_p(f)(B)n iff ∃m',n'. m' B n' and m = p·m' and n = p·n'
Set then P (f) = {P_p(f) | p ε f }.

For each f: AC and p ε f, one has P_p(f) ∈ **ω-Set** [P A,P C] since **ω-Set**[P A,P C] = **Set**[P A,P C] in view of the full realizability relation given to the ω-set P C. (More generally, each set-theoretic function which has as its target an object in the range of Δ: **Set** **ω-Set** is realizable by all indices.)

It is also easy to observe that P (f•g) P (f)•P (g) and id ∈ P (id) for f, g, and id in the due types. This proves that P is a quasi-functor.

We claim that the interpretation of subtyping we are using, faithfully corresponds to the intuitive semantics of subtyping (or is "compelling", as suggested in [Mitchell 88] with reference to [Bruce Longo 89]).

Note first that the coercion $\mathbf{c}_{A,B}$ in general is not a mono (or injective map) in **PER**. It happens to be so only when $Q(A) \subseteq Q(B)$, that is, when one also has $In(A) \leq In(B)$, as ω-sets. Indeed, the topos theoretic notion of subobject as mono from A to B, given by $Q(A) \subseteq Q(B)$, would not be able to give us the antimonotonicity of "" in the first argument, and thus the simple but important theorems 3.4.1 and 3.4.2.

Moreover, in categories (and toposes) one usually works "to within isomorphisms", while the programming understanding of subtypes and inheritance is surely not "to within isomorphism". At most, the programming understanding is "to within *identical* isomorphisms", as a general isomorphism may be a very complicated program and is not likely to be computationally irrelevant.

In conclusion, we want a mathematical semantics which reflects the intuition of the programmer, who views a subtype almost as a subset, but not exactly, as some coercion may be allowed. Our model suggests what sort of coercions may be generally natural: they must be computed by the type-free identical maps and preserved by identical isomorphisms.

This interpretation explains why coercions may disappear in the description of the programming language and why they do not show up at compile time, even though they do not need to be exactly the identity. In our understanding, the compilation of a typed program into its type-free version corresponds to the passage from a morphism in the category of types or kinds, **PER** or **ω-Set**, to its type-free realizers. Type coercions, in particular, are realized by identical computations.

Because of this interplay between sets, computations, and categories, the present approach to subtypes is halfway between the set-theoretic notion of subset and the category (or topos) theoretic subobjects. We claim that this is a suitable mathematical understanding of the programmer's attitude.

We interpret now the formal equivalence of kinds and types as the equality in the model. It is then easy to prove that the relations $\leq$ in 3.2.1-3.2.2, and the quasi-functor P in 3.2.3, satisfy the applicable properties listed under "Kind inclusion" and "Type inclusion", in section 2.9. We are then left with justifying subsumption and coercion, described in section 2.5. We have already discussed the meaning of coercions; these ideas will lead to the formal interpretation of $Quest_C$ in part 4. Subsumption and Quest will be

dealt with in part 5. As already mentioned, recursive types and functions are not considered.

3.3 Operator kinds

The formation, introduction, and elimination rules for operators ([KF P], [TI P], and [TE P]) are easily taken care of. Definition 3.1.6 tells us that we can form a kind, the ω-set $\langle \mathbf{P}_{a\in A} G(a), \varepsilon_{\mathbf{P}} G\rangle$, out of any kind (ω-set) $\langle A, \varepsilon\rangle$ and any function G: A **ω-Set** [KF P]. By definition, the elements of $\langle \mathbf{P}_{a\in A} G(a), \varepsilon_{\mathbf{P}} G\rangle$ are the (computable) functions f such that, when fed with a∈A give as output elements f(a) of G(a). This is exactly what rules [TI P] and [TE P] formalize.

Rule [T P b] is understood in the model by the behavior of a λ-term as a function. Indeed, [T P h] stresses that in any model, functions are interpreted extensionally.

3.4 The kind of types

The lattice **PER** has $\underline{\omega} = (\omega, \omega\omega)$ as largest element, that is, ω with the full relation. Clearly, $\underline{\omega}$ contains just one equivalence class, ω. Thus $\underline{\omega}$ gives meaning to Top, and ω to top. Moreover, the ω-set of all p.e.r.'s is given by $\mathbf{M_0} = \mathsf{P}(\underline{\omega})$.

Rule [TF P] here is given meaning by definition 3.1.7. The interpretation is apparently very simple, but there is a crucial asymmetry with respect to [KF P]. Rule [KF P] has the structure:

$$\frac{\text{kind} \qquad \text{kind}}{\text{kind}}$$

Rule [TF P], instead, looks like:

$$\frac{\text{kind} \qquad \text{type}}{\text{type}}$$

In particular, the kind on the left may be T, the kind of types.

This schema is the crucial type construction in explicit polymorphism. It is impredicative in that, in order to know what types are, one must already know their entire collection, T . ([Feferman 87, 88] and [Longo 88] provide further discussions.) This peculiar type construction is reflected in the related rules.

In [VI P] one allows the formation of terms where abstraction is not done with respect to variables ranging over a type, as in the first-order case. Instead, they range over a kind (possibly T, again). By this, it makes sense by rule [VE P] to apply a term to an element of a kind (possibly a type, and even the type of that very term). This is the dimensional clash which is hard to justify mathematically, and is a central difficulty in the semantics of polymorphism.

Theorem 3.1.8 relates [KF P] and [TF P] by telling us that they are interpreted

by the same construction, in the universe of ω-sets. This gives mathematical unity and clarity of meaning. In particular, it says that the interpretations of terms constructed by [VI Π] are going to be computable functions which may be fed with elements of an ω-set and which then output a term of the expected type, as required by [VE Π] and as modeled in the structure by definition 3.1.6.

Rule [TIncl Π] is validated by the following theorem.

Theorem 3.4.1

Let $\langle A, \Vdash_A \rangle, \langle A', \Vdash_{A'} \rangle \in \omega$**-Set** and G: A **PER**, G': A' **PER**. Assume A'$\leq$A in ω**-Set** and that $\forall a' \in A'$, G(a') $\leq$ G'(a'), in **PER**. Then:

$\Pi_{a \in A} G(a) \leq \Pi_{a' \in A'} G'(a')$, in **PER**.

Proof

Recall that $n\ (\Pi_{a \in A} G(a)_{\mathbf{PER}})\ m$ iff $\forall a \in A.\ \forall p,q \Vdash_A a.\ n \bullet p\ G(a)\ m \bullet q$. Then $\forall a \in A'.\ \forall p,q \Vdash_{A'} a.\ n \bullet p\ G(a)\ m \bullet q$. Since $n \bullet p\ G(a)\ m \bullet q$ implies $n \bullet p\ G'(a)\ m \bullet q$, we are done.

With reference to the discussion on rules [KF Π] and [TF Π] above, a type formation rule for products with the structure:

$$\frac{\text{type} \qquad \text{type}}{\text{type}}$$

would be a first-order rule and may be soundly interpreted over **PER** [Ehrhard 88]. Quest$_{(c)}$ has nothing of this structure for products, as it complicates typechecking and compilation. An implicit use of it is the formal description and the semantics of records given in [Bruce Longo 89]. In the current paper we could avoid any reference to first-order constructs by coding record types in the second-order language (section 2.10). More on their interpretation will be given in section 3.5.

As for ordinary higher type functions, the interpretation of their rules, by corollary 3.1.9, is given as a special case of the meaning of the rules above, except for [TIncl], since in this specific model types happen to be kinds (by the embedding In). The arrow types are just degenerated products (that is, products defined by a constant function, as in 3.1.9).

As an exercise, let's see what happens to the exponents in **PER** and their elements (the equivalence classes). This may be done by a little theorem, which proves the validity of rule [TIncl] in section 2.8.

Proposition 3.4.2

Let A, A',B, B'∈**PER** be such that A' ≤ A and B ≤ B'. Then AB ≤ A'B'. In particular, for n (AB) n, $\ulcorner n \urcorner_{AB} \subseteq \ulcorner n \urcorner_{A'B'}$

Proof

n (AB) m ⇔ ∀p,q. (p A q ⇒ n•p B m•q)

⇒ ∀p,q. (p A' q ⇒ n•p B' m•q),

as p A' q ⇒ p A q ⇒ n•p B m•q ⇒ n•p B' m•q

⇔ n (A'B') m

The rest is obvious.

Proposition 3.4.2 gives the antimonotonicity of in its first argument, as formalized in the rules of Quest [TIncl], and required by inheritance. Moreover, and more related to the specific nature of this interpretation of , proposition 3.4.2 reveals a nice interplay between the extensional meaning of programs and the intensional nature of the underlying structure.

Indeed, typed programs are interpreted as extensional functions in their types, as we identify each morphism in **PER** with the equivalence class of its realizers. That is, if n $\mathbf{r}_{AB}$ f, then $\ulcorner n \urcorner_{AB} \in AB$ represents f∈**PER**[A,B] in the exponent object AB. Assume for example that M: AB is interpreted by f∈**PER**[A,B]. (For the moment we will call A both a type and that type's interpretation as a p.e.r.; see part 4 where the interpretation of terms and types is given.) In the assumption of the proposition, f∈**PER**[A,B] and c(f)∈**PER**[A',B'] are distinct elements, and live in different function spaces. The element c(f) is uniquely obtained by the coercion c, which gives meaning to adjusting the types in M in order to obtain a program in A' B'. Also, when viewed as equivalence classes of realizers, f and c(f) are different sets of numbers.

However, the intended meaning of *inheritance* is that one should be able to run any program in AB on terms of type A' also, as A' is included in A. When n $\mathbf{r}_{AB}$ f, this is exactly what $\ulcorner n \urcorner_{AB} \subseteq \ulcorner n \urcorner_{A'B'}$ expresses: any computation which realizes f in the underlying type-free universe actually computes c(f) also. Of course, there may be more programs for c(f), in particular if A' is strictly smaller than A. Thus, even though f and c(f) are distinct maps (at least because they have different types) and interpret different programs, their type-free computations are related by a meaningful inclusion, namely $\ulcorner n \urcorner_{AB} \subseteq \ulcorner n \urcorner_{A'B'}$ in this model.

This elegant interplay between the *extensional collapse,* which is the key step in the hereditary construction of the types as partial equivalence relations, and the intensional nature of computations is a fundamental

feature of the realizability models.

3.5 Records

Formally, there is nothing to be said about the semantics of records, as they are a derived notion. However, we mention one crucial merit of the coding proposed and its meaning.

Record types should not be understood simply as cartesian products. The main reason is that the meaning of a record type R' with *more* fields than a record type R (but where all the fields in R are in R') should be *smaller* than the meaning of R. Indeed, R' contains *fewer* record relizers. This situation was obtained, say, in the **PER** interpretation of [Bruce Longo 89] by understanding record types as indexed, first-order products. That is, if I is a (finite) set of (semantic) labels, then $\mathbf{P}_{i\in I}A_i$ would interpret a record whose fields are interpreted by the A_i 's. By theorem 3.4.1, $\mathbf{P}_{i\in I}A_i$ gives the required contravariance in the meaning of records.

In the present approach, we can use the expressive power of Quest as a higher-order language with a Top type, and model records with little effort. Record types are coded as ordered tuples. Top is the last factor of the product and replaces missing fields (with respect to the order), and by doing so it guarantees contravariance. This intuition is precisely reflected in the model, by interpreting Top as the largest p.e.r.. Thus, any extension of a given record type by informative fields, that is, by fields whose meaning is different from the full relation on ω, gives smaller p.e.r.'s.

4. Semantic interpretation of Quest_C

In this section we give the formal semantics of Quest_C over the **ω-Set/PER** model. The basic idea, for the inductive definition, is to interpret type environments as ω-sets with a realizability notion which codes pairs as elements of a dependent sum. In this way, if for example E = (∅, y: B, x: A), then ⟦E⟧ contains all pairs:

<e,a> with e∈⟦∅, y: B⟧ and a∈⟦∅, y: B ∫ A type⟧e

In this approach one has to interpret judgments, not just terms, as judgments contain the required information to interpret (free) variables. For example, the variable x is given meaning within the judgment E ∫ x:A, say, for E as above. In particular, its interpretation ⟦E ∫ x:A⟧e', for a fixed environment value e' = <e,a>∈⟦E⟧, is the second projection and gives a∈⟦∅, y: B ∫ A type⟧e. (See also [Scedrov 1988], [Luo 1988].) The projection is clearly a realizable map, that is, it is computed by the index of a partial recursive function. Note that the interpretation of closed terms depends on the judgments they appear in, in particular on the types they are assigned to.

Moreover, the meaning of a judgment gives, simultaneously, the interpretation of a construct (kind, type, or term) and makes a validity assertion; for example, it says that a given term actually lives in the given type, under the given assumptions.

Kinds, types, and terms are interpreted as maps from the ω-set interpreting the given environment to **ω-Set**, **PER**, and the intended type, respectively. As our morphisms are extensional functions, the interpretation is uniquely determined by their behavior on the elements of the environment. The indexes realizing these maps may be computed by induction, using as base the indexes for the projection functions. The crucial step is the interpretation of lambda abstraction and application for terms. For example, given a realizer p for the map <e,A> ∈ ⟦E,X::K ∫ b:B⟧<e,A>, a realizer for e ∈ ⟦E ∫ Π(X::K)b : Π(X::K)B⟧e is obtained by the recursive function s of the s-m-n (or iteration) theorem, namely by an index for n ∈ s(<p,n>), where s(<p,n>)(m) = p(<n,m>). Similarly, any index for the universal partial recursive function gives the realizers for an applicative term. We prefer to leave to the reader the intensional details of the computations and focus on the extensional presentation of the interpretation maps. These maps already require a fair amount of detail for a full description and should not be further obscured by the explicit mention of the indexes of the realizable functions.

Observe that, in a fixed environment, kinds are interpreted as ω-sets, while types are p.e.r.'s. More precisely, operator kinds are functions which take an element of a kind (possibly a type) as input and give an element of a kind (possibly a type) as output. Also, these functions live in an ω-set, which is obtained as an indexed product in the sense of 3.1.6.

As is common when dealing with CCC's, we make no distinction between an exponent object, the p.e.r. AB, say, and the set of morphisms, **PER**[A,B], it represents. Thus, the meaning of a term in **PER**[A,B], say, may be viewed either as a function from the p.e.r. A to the p.e.r. B, or as the equivalence class of its realizers in the p.e.r. AB (see also definition 4.1.1.(1) below). This poses no problem with regard to **ω-Set**, since an exponent object is exactly an (ω-)set of (realizable) functions, as in the category of sets.

4.1 Interpretation

We interpret, in order, environments, kinds, types, and terms.

Environments

$E = \varnothing$ $\qquad$ $\llbracket E \rrbracket = \langle \{1\}, \varepsilon \rangle$ $\qquad$ where $\qquad$ $\forall n \in \omega\ n \varepsilon 1$

E = E', X::K $[\![E]\!] = \langle\{\langle e,A\rangle \mid e\in[\![E']\!] \wedge A\in[\![E' \vdash K\ \mathrm{kind}]\!]e\},\ \varepsilon_E\rangle$
where $\langle n,m\rangle\ \varepsilon_E\ \langle e,A\rangle$ iff $n\ \varepsilon_{E'}\ e$ and $m\ \varepsilon_{[\![E' \vdash K\ \mathrm{kind}]\!]e}\ A$

E = E', x:A $[\![E]\!] = \langle\{\langle e,a\rangle \mid e\in[\![E']\!] \wedge a\in[\![E' \vdash A\ \mathrm{type}]\!]e\},\ \varepsilon_E\rangle$
where $\langle n,m\rangle\ \varepsilon_E\ \langle e,a\rangle$ iff $n\ \varepsilon_{E'}\ e$ and $m\ \varepsilon_{[\![E' \vdash A\ \mathrm{type}]\!]e}\ a$

Kinds

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash \top\ \mathrm{kind}]\!]e = \mathcal{M}_0$

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash \mathcal{P}(A)\ \mathrm{kind}]\!]e = \mathcal{P}\,[\![E \vdash A\ \mathrm{type}]\!]e$

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash \mathcal{P}(X::K)L\ \mathrm{kind}]\!]e = \langle \mathcal{P}_{A\in[\![E\vdash K\ \mathrm{kind}]\!]e} G(A), \varepsilon_{\mathcal{P}G}\rangle$
where G: $[\![E \vdash K\ \mathrm{kind}]\!]e$ **ω-Set** is given by
$G(A) = [\![E,X::K \vdash L\ \mathrm{kind}]\!]\langle e,A\rangle$

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash \lambda(X::K)B :: \mathcal{P}(X::K)L]\!]e \in \mathcal{P}_{A\in[\![E\vdash K\ \mathrm{kind}]\!]e}[\![E,X::K\vdash L\ \mathrm{kind}]\!]\langle e,A\rangle$
such that $\forall A\in[\![E \vdash K\ \mathrm{kind}]\!]e.$
$([\![E \vdash \lambda(X::K)B :: \mathcal{P}(X::K)L]\!]e)(A) = [\![E,X::K \vdash B::L]\!]\langle e,A\rangle$

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash B(A) :: L\{X\leftarrow A\}]\!]e = ([\![E \vdash B:: \mathcal{P}(X::K)L]\!]e)([\![E \vdash A::K]\!]e)$

Types

$\vdash$ E = E', X_n::K_n, E'' $\forall e = \langle\ldots\langle e_n,A_n\rangle,\ldots\rangle\in[\![E]\!].\ [\![E \vdash X_n::K_n]\!]e = A_n\in [\![E' \vdash K_n\ \mathrm{kind}]\!]e_n$

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash \mathrm{Top}\ \mathrm{type}]\!]e = \underline{\omega} = (\omega, \omega\omega)$

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash \mathcal{P}(X::K)B\ \mathrm{type}]\!]e = \mathcal{P}_{A\in[\![E\vdash K\ \mathrm{kind}]\!]e}[\![E,X::K \vdash B\ \mathrm{type}]\!]\langle e,A\rangle$

$\vdash$ E env $\forall e\in[\![E]\!].\ [\![E \vdash AB\ \mathrm{type}]\!]e = [\![E \vdash A\ \mathrm{type}]\!]e\ [\![E \vdash B\ \mathrm{type}]\!]e$

Terms

E = E', x_n:A_n, E'' $\forall e = \langle\ldots\langle e_n,a_n\rangle,\ldots\rangle\in[\![E]\!].\ [\![E \vdash x_n:A_n]\!]e = a_n\in [\![E' \vdash A_n\ \mathrm{type}]\!]e_n$

$\int$ E env $\forall e \in [\![E]\!].\ [\![E \int top:Top]\!]e = \omega$

$\int$ E env $\forall e \in [\![E]\!].\ [\![E \int c_{A,B}(a):B]\!]e = c_{[\![E \int A\ type]\!]e,[\![E \int B\ type]\!]e}([\![E \int a:A]\!]e)$

$\int$ E env $\forall e \in [\![E]\!].\ [\![E \int \lambda(X::K)b : P(X::K)B]\!]e$
$\in P_{A \in [\![E \int K\ kind]\!]e}[\![E,X::K \int B\ type]\!]<e,A>$
such that $\forall A \in [\![E \int K\ kind]\!]e.$
$([\![E \int \lambda(X::K)b : P(X::K)B]\!]e)(A) = [\![E,X::K \int b:B]\!]<e,A>$

$\int$ E env $\forall e \in [\![E]\!].\ [\![E \int b(A) : B\{X \leftarrow A\}]\!]e = ([\![E \int b : P(X::K)B]\!]e)([\![E \int A\ type]\!]e)$

$\int$ E env $\forall e \in [\![E]\!].\ [\![E \int \lambda(x:A)b : AB]\!]e \in [\![E \int A\ type]\!]e[\![E \int B\ type]\!]e$
such that $\forall a \in [\![E \int A\ type]\!]e.$
$([\![E \int \lambda(x:A)b : AB]\!]e)(a) = [\![E,x:A \int b:B]\!]<e,a>$

$\int$ E env $\forall e \in [\![E]\!].\ [\![E \int b(a) : B]\!]e = ([\![E \int b : AB]\!]e)([\![E \int a:A]\!]e)$

In view of the interpretation of kinds, types, and terms, the meaning of the judgments is the obvious one. The :: and : relations go to $\in$ for ω-sets and p.e.r.'s, respectively; the relations <:: and <: are interpreted as subkind and subtype in **ω-Set** and **PER**; finally, <::> and <:> are just equality.

Indeed, by induction on types and terms, one may check directly that this is a good interpretation. In particular, one can check that all the given functions are actually realized, as mentioned above, and hence that types and terms inhabit the intended function and product spaces; see 4.1.2. (For example, $[\![E \int \lambda(X::K)b : P(X::K)B]\!]e$ is actually in $P_{A \in [\![E \int K\ kind]\!]e}[\![E,X::K \int B\ type]\!]<e,A>$.) However, this also follows from general categorical facts, namely the cartesian closure of **ω-Set** and the observation that **PER**, viewed as $\mathbf{M}_0$, is an internal CCC of **ω-Set** where the internal product P is right adjoint to the diagonal functor. (We obtain an *internal model* of Girard's Fω; see [Asperti Longo 1990] where the general categorical meaning of Fω is given.)

The next theorem, whose proof is left to the reader, summarizes all these facts, and states the soundness of the interpretation. Before stating it, though, we set a better foundation for the interplay of the interpretations of "terms as functions" and "terms as equivalence classes". This is done by the following definition which extends the applicative structure of $(\omega, \cdot)$ to

equivalence classes, and also to the application of an equivalence class to an element of an ω-set (cf. 3.1.7).

Definition 4.1.1

1 - Let A and B be p.e.r.'s. Define then, for n(AB)n and mAm,

$\ulcorner n \urcorner_{AB} \cdot \ulcorner m \urcorner_A = \ulcorner n \cdot m \urcorner_B$

2 - Let $\langle K, \Vdash_K \rangle \in \omega$**-Set** and G: K **PER**. Set, for short, $\mathbf{P} = \mathbf{P}_{A \in K} G(A)_{\mathbf{PER}}$ and define, for n$\mathbf{P}$n, A∈K, and p $\Vdash$ A:

$\ulcorner n \urcorner_{\mathbf{P}} \cdot A = \ulcorner n \cdot p \urcorner_{G(A)}$

(Note that "$\cdot$" : $\mathbf{P} \times K_{A \in K} G(A)$ depends on K and G.) This is well defined as $\ulcorner n \cdot p \urcorner_{G(A)}$ does not depend on the choice of the number p, which realizes A.

By this explicit reconstruction of the applicative behavior, one may more clearly understand equivalence classes in the p.e.r.'s AB and $\mathbf{P}_{AK} G(A)_{\mathbf{PER}}$ as functions in the due types.

Theorem 4.1.2

$\int$ E env	⇒	$[\![E]\!]$ is a well-defined ω-set
E $\int$ K kind	⇒	∀e∈$[\![E]\!]$. $[\![E \int K \text{ kind}]\!]e$ is a well-defined ω-set
E $\int$ A::K	⇒	∀e∈$[\![E]\!]$. $[\![E \int A::K]\!]e \in [\![E \int K \text{ kind}]\!]e$
E $\int$ A type	⇒	∀e∈$[\![E]\!]$. $[\![E \int A \text{ type}]\!]e \in \mathbf{M}_0$
E $\int$ a:A	⇒	∀e∈$[\![E]\!]$. $[\![E \int a:A]\!]e \in [\![E \int A \text{ type}]\!]e$
E $\int$ K <:: L	⇒	∀e∈$[\![E]\!]$. $[\![E \int K \text{ kind}]\!]e \leq [\![E \int L \text{ kind}]\!]e$ in ω**-Set**
E $\int$ A <: B	⇒	∀e∈$[\![E]\!]$. $[\![E \int A \text{ type}]\!]e \leq [\![E \int B \text{ type}]\!]e$ in **PER**
E $\int$ K <::> L	⇒	∀e∈$[\![E]\!]$. $[\![E \int K \text{ kind}]\!]e = [\![E \int L \text{ kind}]\!]e$
E $\int$ A <:> B	⇒	∀e∈$[\![E]\!]$. $[\![E \int A \text{ type}]\!]e = [\![E \int B \text{ type}]\!]e$
E $\int$ a ↔ b	⇒	∀e∈$[\![E]\!]$. $[\![E \int a:A]\!]e = [\![E \int b:A]\!]e$

4.2 Emulating coercions by bounded quantification

In Quest_C and in its current interpretation we have no subsumption, but instead we have coercions. This means that programs of the form

$(\lambda(x:B)d)(a)$ where a:A<:B (with A≠B) (1)

are not legal: an explicit coercion has to be applied, as in

$(\lambda(x:B)d)(c_{A,B}(a))$ (2)

In this latter case, one may avoid both subsumption and coercions and

recast (1) via an additional bounded quantifier:

$(\lambda(X<:B)\lambda(x:X)d)(A)(a)$ (3)

It is clear that (3) has the same effect as (1) or as (2), since this is how (1) can be correctly expressed in our current framework, by coercions. The fact that (2) and (3) are equivalent is a fairly deep property of the semantics, relating a bounded quantifier to a coercion. In general, this is not derivable from the syntax.

The following theorem states that, semantically, coercions can be removed in favor of bounded quantifiers.

Recall that $E \int a : A \wedge E \int A <: B \Rightarrow E \int c_{A,B}(a) : B$.

Theorem 4.2.1

Assume that $E \int d : D$, $E \int a : A$ and $E \int A <: B$. Then, in **PER** one has

$(\lambda(X<:B)\lambda(x:X)d)(A)(a) = (\lambda(x:B)d)(c_{A,B}(a))$

Proof

For simplicity, we fix an environment e and identify types A, B, and D with their meanings as p.e.r. in e.

Set $P = P_{X\leq B}XD$ and let $\ulcorner n \urcorner_P = [\![E \int (\lambda(X<:B)\lambda(x:X)d) : P(X<:B)(XD)]\!]e \in P$. Then $\ulcorner n \urcorner_P \dagger C = \ulcorner n \cdot p \urcorner_{CD}$ for any C, such that $E \int C <: B$, and any p, since any number p realizes C, when $C \leq B$, by definition of the power quasi-functor.

Let now m be such that $[\![E \int a : A]\!]e = \ulcorner m \urcorner_A$. Then $c_{A,B}(\ulcorner m \urcorner_A) = \ulcorner m \urcorner_B$ and:

$[\![E \int (\lambda(X<:B)\lambda(x:X)d)(A)(a) : D]\!]e$
$= \ulcorner n \urcorner_P \dagger A \dagger \ulcorner m \urcorner_A = \ulcorner n \cdot p \urcorner_{AD} \dagger \ulcorner m \urcorner_A = \ulcorner n \cdot p \cdot m \urcorner_D$
$= \ulcorner n \cdot p \urcorner_{BD} \dagger \ulcorner m \urcorner_B$ where $n \cdot p(BD) n \cdot p$ by 4.1.1(1)
$= \ulcorner n \urcorner_P \dagger B \dagger \ulcorner m \urcorner_B$ by 4.1.1(2)
$= [\![E \int (\lambda(X<:B)\lambda(x:X)d)(B)c_{A,B}(a) : D]\!]e$
$= [\![E \int (\lambda(x:B)d)c_{A,B}(a) : D]\!]e$ by the syntax.

In $Quest_C$, we dropped the subsumption rule in favor of coercions. However, there is also a proof-theoretic reason to warn the programmer about the use of subsumption in connection with (η); namely, the equational system of typed terms would not be Church-Rosser any more (with respect to the obvious reduction rules). Consider say:

$\lambda(x:A)(\lambda(y:B)e)x$ (with $x \notin FV(\lambda(y:B)e)$)

where x is not free in $\lambda(y:B)e$, and let $A <: B$.

In the presence of subsumption, this program would type-check, for any e and C such that e:C. However,

$\lambda(x:A)(\lambda(y:B)e)x \P \lambda(y:B)e : B \to C$ by (η)

λ(x:A)(λ(y:B)e)x ¶ λ(x:A)e : A → C by (β)

and confluence would be lost. Because of this, we abandon (η) in part 5.

In $Quest_C$, the program one has in mind when writing λ(x:A)(λ(y:B)e)x, is actually described by the polymorphic term:

λ(x:A)(λ(X<:B) λ(y:X) e)(A)(x)

which yields confluent reductions.

For this reason, (η) is adopted in Quest as an equality rule, but not as a computation rule.

5. Semantic interpretation of Quest

In this section we model the original version of Quest, namely the language based on the subsumption rule [TSub / Quest] of section 2.9, instead of on coercions.

Subsumption is important for at least two reasons. First, programming with explicit coercions becomes too cumbersome; much of the appeal of subtyping has to do with the flexibility and compactness provided by subsumption. Second, subsumption is intended not as an arbitrary coercion, but as a coercion that performs no work; this is essential for capturing the flavor of object-oriented programming, where subsumption is used freely as a way of viewing objects as members of different types.

Hence we feel we are justified in presenting more complex semantic techniques in order to give a faithful representation of subsumption.

Let $(D, \cdot)$ be a model of type-free lambda calculus. The construction of the categories **D-Set** and $\mathbf{PER_D}$ over $(D, \cdot)$ works similarly. Indeed, all the work carried on so far can be easily generalized to any (possibly partial) Combinatory Algebra or model of Combinatory Logic. In view of the relevance of Kleene's realizability interpretation of Intuitionistic Logic for these models, it is fair to call "realizability structures" the categories **D-Set** and $\mathbf{PER_D}$ over a Combinatory Algebra $(D, \cdot)$. As already mentioned, we preferred $(\omega, {}^{\dagger})$ as it is more directly related to Kleene's work and because of the immediate intuitive appeal of classical recursion theory. However, we now need to be able to give meaning to type-free terms, which cannot be done over $(\omega, {}^{\dagger})$. For this purpose, we work over an arbitrary λ-model: that is, an applicative structure $(D, \cdot)$ with an interpretation $D[\![-]\!]$ of λ-terms defined, say, as in [Hindley Longo 80] or [Barendregt 84].

The interpretation of Quest is given in two steps. First we translate typed terms into terms of the type-free calculus, by "erasing-types". We add to the latter only a constant symbol "top", in order to take care of the

corresponding constant in Quest.

In the second step, we use the meaning of the erased terms to interpret typed terms. Environments, kinds, and types will be interpreted as in $Quest_C$, except for an "isomorphic change" in the interpretation of product types. As for types in particular, this interpretation is possible since, in view of our formal definition of subkinds and of its semantics, we had no kind coercions even in $Quest_C$, but just type coercions.

Terms may still be understood as morphisms, in the due types. We already used the identification of morphisms with the equivalence classes of their realizers. In the interpretation of Quest we exploit this correspondence and interpret typed terms directly as equivalence classes, with no ambiguity.

Briefly, for each environment $e = \langle ...\langle e_n, a_n\rangle,...\rangle \in [\![E]\!]$ we choose an environment map s_e: Var $\to$ D which picks up an element of the equivalence class a_n. Then, by using these environment maps, we interpret a typed term as the equivalence class which contains the interpretation of its erasure.

The interpretation will not depend on the particular choice of the environment map.

5.1 Preliminaries and structures

The categories **D-Set** and $\mathbf{PER_D}$ over $(D, \cdot)$ are defined exactly as **ω-Set** and $\mathbf{PER_\omega}$ over $(\omega, \cdot)$, in 3.1.1 and 3.1.2. However, their use in the semantics of Quest will be slightly changed in a crucial point. Second-order impredicative quantification will not be interpreted exactly by the set-theoretic indexed product of realizable functions, as in 3.1.7. We will use instead an isomorphic, but not identical, interpretation of this quantification by p.e.r.'s obtained as a straightforward set-theoretic intersection. This is made possible by the following simple, but fundamental theorem, which establishes a connection between the previous interpretation of higher-order quantification and the one given in [Girard 72] and [Troelstra 73]. It was first suggested by Moggi and actually started most of the recent work on the semantics of polymorphism, by suggesting that Girard's model could be given a relevant categorical explanation. (See remark 3.1.5.) We use it here as a tool for our semantic interpretation of Quest. We report its proof since it matters for our purposes, as we point out in remark 5.1.2. Note first that, if $\{A_i\}_{i\in I}$ is a collection of p.e.r.'s, then $\bigcap_{i\in I} A_i$ is also a p.e.r. by

$$n(\bigcap_{i\in I} A_i)m \text{ iff } n\ A_i\ m \text{ for all } i\in I$$

Theorem 5.1.1

Let $\langle A, \varepsilon_A\rangle \in$ **D-Set** be such that $\varepsilon_A = D\times A$ and let G: A $\to$ $\mathbf{PER_D}$. Then:

$$(\prod_{a\in A} G(a))_{\mathbf{PER_D}} \cong \bigcap_{a\in A} G(a) \text{ in } \mathbf{PER_D}.$$

Proof ([Longo Moggi 88])

Let $S = {}_{a\in A}G(a) \in \mathbf{PER_D}$. By definition both $\mathbf{P}_{a\in A}G(a)_{\mathbf{PER_D}}$ and S are in $\mathbf{PER_D}$. Thus we need to define a bijection $H: S \to \mathbf{P}_{a\in A}G(a)_{\mathbf{PER_D}}$ and prove that it is realized with its inverse.

Let $H(\ulcorner n \urcorner_S) = \lambda a\in A.\ulcorner n \urcorner_{G(a)}$. Clearly, $H(\ulcorner n \urcorner_S) \in \mathbf{P}_{a\in A}G(a)$ and H is well defined, since $\ulcorner n \urcorner_S = \ulcorner m \urcorner_S$ implies, n G(a) m for all $a\in A$, and hence $\ulcorner n \urcorner_{G(a)} = \ulcorner m \urcorner_{G(a)}$.

Consider now the combinator k such that $k\cdot p\cdot q = p$, for all $p, q \in \mathsf{D}$. Then $k\cdot n$ realizes $H(\ulcorner n \urcorner_S)$, since

$$\forall a\in A.\ \forall q \Vdash_A a.\ k\cdot n\cdot q = n \in \ulcorner n \urcorner_{G(a)} = H(\ulcorner n \urcorner_S)(a),$$

and k realizes H. It is easy to observe that H is injective. Let us prove that H is surjective.

If $h \in \mathbf{P}_{a\in A}G(a)$, then by definition, $\exists m \Vdash_{\mathbf{P}G} h$; that is,

$$\exists m.\ \forall a\in A.\ \forall q \Vdash_A a.\ m\cdot q \Vdash_{G(a)} h(a) \qquad \text{or, equivalently,}$$

$$\exists m.\ \forall a\in A.\ \forall q\in \mathsf{D}.\ h(a) = \ulcorner m\cdot q \urcorner_{G(a)}, \text{ as } \Vdash_A = \mathsf{D}\times A$$

Fix now an element 0 of D. Then, for $n = m\cdot 0$, we have $\forall a\in A.$ n G(a) n, that is, n S n. In conclusion, $\forall a\in A.\ H(\ulcorner n \urcorner_S)(a) = \ulcorner n \urcorner_{G(a)} = h(a)$, that is, $H(\ulcorner n \urcorner_S) = h$. Therefore H^{-1} exists and it is realized by any $p \in \mathsf{D}$ such that $p\cdot m = m\cdot 0$, for all $m \in \mathsf{D}$.

Remark 5.1.2

The key idea in the proof consists in defining the applicative or functional behavior of each equivalence class $\ulcorner n \urcorner_S$, say, in $S = {}_{a\in A}G(a) \in \mathbf{PER_D}$, by setting

$$\ulcorner n \urcorner_S \dagger a = \ulcorner n \urcorner_{G(a)}$$

This is how, to within isomorphism, $\ulcorner n \urcorner_S$ defines a function in $\mathbf{P}_{a\in A}G(a)$. Observe that, when the isomorphism is given by the "constant-constructor" combinator k, the proof relates this notion of application to the application $\ulcorner n \urcorner_{\mathbf{P}} \dagger a = \ulcorner n\cdot p \urcorner_{G(a)}$, for $p \Vdash_A a$, as defined in 4.1.1. Indeed, $\ulcorner n\cdot p \urcorner_{G(a)}$ is constant with respect to p, under the assumption $\Vdash_A = \mathsf{D}\times A$ in 5.1.1. The next proposition shows that this assumption is satisfied by the D-sets we are interested in: that is, by the definable ones, in the language of Quest.

Proposition 5.1.3

Let $\int$ E env and E $\int$ K kind. Then, for all $e\in[\![E]\!]$, $[\![E \int K \text{ kind}]\!]e$ is a D-set $\langle \underline{A}, \Vdash_{\underline{A}} \rangle$ with $\Vdash_{\underline{A}} = \mathsf{D}\times\underline{A}$.

Proof

This is clearly true for the base of the induction, in view of the interpretation of T and P (C), for any type C. (Recall that one even has T = P (Top)). Consider now E $\int$ P(X::K)L kind. Then:

$$\forall e\in[\![E]\!].\ [\![E \int \mathbf{P}(X::K)L \text{ kind}]\!]e = \langle \mathbf{P}_{A\in[\![E\int K \text{ kind}]\!]e}[\![E,X::K\int L$$

kind⟧<e,A>, ε_{P_G}⟩,
where G(A) = ⟦E,X::K∫L kind⟧<e,A>. By induction, just assume that, for all e and A, the D-set L(e,A) = ⟦E,X::K∫L kind⟧<e,A> has the full ε_L relation. Then any set theoretic function f in $\times_{A\in[\![E\int K\ kind]\!]e}$⟦E,X::K∫L kind⟧<e,A> is realized by any n ∈ D, since one always has n·pε_L f(A), no matter which A∈⟦E∫ K kind⟧e and p are taken.

Remark 5.1.4 (For readers with some experience in Category Theory.)

Continuing from remark 3.1.5. In [Hyland 87] and [Longo Moggi 88], the existence of a (internal) right adjoint to the diagonal functor, that is, the small completeness of **PER** in the Effective Topos or in **ω-Set**, is shown by taking exactly the intersection as product (see [Asperti Longo 90] for details). This fully justifies the interpretation below of second-order impredicative types as intersections.

5.2 Interpretation [-]'

We now translate typed terms into terms of the type-free calculus, by erasing all type information. The type-free λ-calculus is extended by a constant symbol, top.

Definition 5.2.1

The translation map *erase* from typed terms into type-free terms is defined by induction on the structure of terms:

erase(x) = x
erase(top) = top
erase(λ(x:A)b) = λx. *erase*(b)
erase(b(a)) = *erase*(b)*erase*(a)
erase(λ(X::K)b) = *erase*(b)
erase(b(A)) = *erase*(b)

With the preliminaries above, it is now straightforward to implement our idea: a typed term is interpreted by the equivalence class of its erasure, with respect to its type as p.e.r.. We then need to show that this interpretation is sound. Indeed, this interpretation generalizes a theorem stated in [Mitchell 86] and tidily relates to the alternative approach to the semantics of the subsumption rule [TSub / Quest] in [Bruce Longo 89]. Observe that this interpretation, in contrast to the early attempt in [Bruce Longo 89], is direct. This is made possible by the use of theorem 5.1.1, since by erasure the meaning of a second-order typed term becomes an element of the

intersection of all the types which form its range. For example, the polymorphic identity function $\lambda(X::T)\,\lambda(x:X)\,x : \Pi(X::T)\,(X\to X)$ will be interpreted as the equivalence class of the type-free identity $\lambda x.x$, which happens to live in $A\to A$, for any type A.

Note finally that, since the interpretations of type-free terms are elements of D, while the elements of types as p.e.r.'s are equivalence classes, we need a choice map to obtain an environment for type-free terms from an environment for typed ones. This is done by the following definition.

Definition 5.2.2

Given $E = E', x_n:A_n, E''$ and $e = \langle\ldots\langle e_n,a_n\rangle,\ldots\rangle \in [\![E]\!]$, fix s_e: Var $\to D$ such that $s_e(x_n) \in a_n \in [\![E' \vdash A_n\ \mathrm{type}]\!]'e_n$, where $[\![E]\!]$ is defined as in section 4.1, and $[\![E' \vdash A_n\ \mathrm{type}]\!]'e_n$ is the interpretation of types given below.

Note that s_e is defined only on term variables and gives no meaning to X::K. The interpretation below will not depend on the choice of s_e. Recall that $D[\![-]\!]$ is the interpretation of type-free terms in $(D, \cdot)$.

Environments

$[\![E]\!]'$ coincides with $[\![E]\!]$ for Quest_c

Kinds No change.

Types No change, except for:

$\vdash E\ \mathrm{env} \qquad \forall e\in[\![E]\!].\ [\![E \vdash \Pi(X::K)B\ \ \mathrm{type}]\!]'e = {}_{A\in[\![E\vdash K\ \mathrm{kind}]\!]'e}[\![E,X::K\vdash B\ \mathrm{type}]\!]'\langle e,A\rangle$

Terms

$\vdash E\ \mathrm{env} \qquad \forall e\in[\![E]\!]'.\ [\![E \vdash a : A]\!]'e = \text{“}\ D[\![\mathit{erase}(a)]\!]s_e\text{”}\ [\![E \vdash A\ \mathrm{type}]\!]'e$

Since higher-order quantification is interpreted as intersection, by an even easier proof than for Quest_c, we have:

Lemma 5.2.3

$E \vdash A<:B$ implies $\forall e\in[\![E]\!]'.\ [\![E \vdash A\ \mathrm{type}]\!]'e \subseteq [\![E \vdash B\ \mathrm{type}]\!]'e$

The following theorem proves the soundness of the interpretation.

Proposition 5.2.4

The interpretation $[\ ]'$ is a well-defined meaning for kinds, types, and terms over **D-Set** and $\mathbf{PER_D}$.

Proof

We need to check only the result for terms, since kinds pose no problem, and there has been enough discussion concerning types and the use of intersection as product. Recall from proposition 5.1.3 that $[E \vdash K\ \mathrm{kind}]'e$ is a D-set with the full relation.

Thus we show by induction on the derivation that, for each $E \vdash a : A$, $D[\mathrm{erase}(a)]s_e$ is in the domain of $[E \vdash A\ \mathrm{type}]'e$ and that it has the correct functional behavior.

Case $E = E', x_n{:}A_n, E'' \vdash x_n{:}A_n$

$\vdash E\ \mathrm{env}\ \ \forall e \in [E]'.\ [E \vdash x_n{:}A_n]'e = \text{“}s_e(x_n)\text{”}\,[E \vdash A_n\ \mathrm{type}]'e_n$

which corresponds to

$\forall e = \langle \ldots \langle e_n, a_n \rangle, \ldots \rangle \in [E].\ [E \vdash x_n{:}A_n]'e = a_n \in [E \vdash A_n\ \mathrm{type}]'e_n$

Case $E \vdash \mathrm{top}{:}\mathrm{Top}$

Just recall that ω is the only element of $\underline{\omega}$.

Case $E \vdash b(a) : B$

$\forall e \in [E]'.\ [E \vdash b(a) : B]'e = \text{“}D[\mathrm{erase}(ba)]s_e\text{”}\,[E \vdash B\ \mathrm{type}]'e$

$= \text{“}D[\mathrm{erase}(b)\mathrm{erase}(a)]s_e\text{”}\,[E \vdash B\ \mathrm{type}]'e$

$= (\text{“}D[\mathrm{erase}(b)]s_e\text{”}\,[E \vdash AB\ \mathrm{type}]'e) \cdot (\text{“}D[\mathrm{erase}(a)]s_e\text{”}\,[E \vdash A\ \mathrm{type}]'e)$

$= ([E \vdash b : AB]'e) \cdot ([E \vdash a{:}A]'e)$

where application between equivalence classes is defined as in 4.1.1.

This simultaneously proves that $[\ -\ ]'$ decomposes soundly and that $D[\mathrm{erase}(ba)]s_e$ is in $\mathrm{dom}([E \vdash B\ \mathrm{type}]'e)$.

Case $E \vdash \lambda(x{:}A)b : AB$

$\forall e \in [E]'.\ [E \vdash \lambda(x{:}A)b : AB]'e = \text{“}D[\lambda x.\mathrm{erase}(b)]s_e\text{”}\,[E \vdash AB\ \mathrm{type}]'e$

which is well defined because by induction, from the semantics of $E, x{:}A \vdash b : B$, one has for all $n \in D$:

$n\ ([E \vdash A\ \mathrm{type}]'e)\ n \ \Rightarrow\ (D[\mathrm{erase}(b)]s_e[n/x])$ is in $\mathrm{dom}([E \vdash B\ \mathrm{type}]'e)$

Thus $D[\lambda x.\mathrm{erase}(b)]s_e$ is in $\mathrm{dom}([E \vdash AB\ \mathrm{type}]'e)$, by virtue of the familiar substitution lemmas in the type-free model $(D, \cdot, D[\ -\ ])$. (See [Barendregt 84].)

Case $E \vdash \lambda(X{::}K)b : \Pi(X{::}K)B$

$\forall e \in [E]'.\ [E \vdash \lambda(X{::}K)b : \Pi(X{::}K)B]'e = \text{“}D[\mathrm{erase}(b)]s_e\text{”}_{\Sigma}$

where $\Sigma = \cap_{A::K}\{[E \vdash B\{X \leftarrow A\}\ \mathrm{type}]'e\}$. (Note that, by the usual substitution

techniques, one has ⟦E ⊢ B{X←A} type⟧'e = ⟦E,X::K ⊢ B type⟧'<e,A>, where we keep identifying the semantic and the syntactic type A by an abuse of language.) This is well defined just as before, since, by induction, one has:

E,X::K ⊢ b : B implies D⟦erase(b)⟧s_e is in dom(⟦E,X::K ⊢ B type⟧'e)

However, in contrast to the previous case, D⟦erase(b)⟧s_e does not depend on X::K while B and its semantics do. Exactly because of this, for all types A one has

D⟦erase(b)⟧s_e is in dom(⟦E ⊢ B{X←A} type⟧'e)

and thus D⟦erase(b)⟧s_e is in dom(Σ). The next case describes also the applicative behavior of ⟦E ⊢ ι(X::K)b : Π(X::K)B⟧'e.

Case E ⊢ c(A) : B{X←A}

∀e∈⟦E⟧'. ⟦E ⊢ c(A) : B{X←A}⟧'e = "D⟦erase(c)⟧s_e"⟦E ⊢ B{X←A} type⟧'e

by the definition of erase. Observe now that one must have E ⊢ c : Π(X::K)B. By setting

Σ = ∩$_{A::K}$⟦E ⊢ B{X←A} type⟧'e}

by the previous case and the definition of erase, one has

∀e∈⟦E⟧'. ⟦E ⊢ c : Π(X::K)B⟧'e = "D⟦erase(c)⟧s_e"$_Σ$ in dom(Σ)

Thus, for all A D⟦erase(c)⟧s_e is in dom(⟦E ⊢ B{X←A} type⟧'e).

By this and by the definition of application of an intersection class to a p.e.r., given in 5.1.2, compute

D⟦erase(c)⟧s_e"⟦E ⊢ B{X←A} type⟧'e = ("D⟦erase(c)⟧s_e"ς) · (⟦E ⊢ A type⟧'e)

= (⟦E ⊢ c:Π(X::K)B⟧'e) · (⟦E ⊢ A type⟧'e)

We have also proved:

Corollary 5.2.5

If ⊢ E env, then ∀e∈⟦E⟧. ⟦E ⊢ a : A⟧'e ∈ ⟦E ⊢ A type⟧'e.

It is a minor variant of the work done for Quest$_c$ to check fully that we provided an interpretation for Quest (that is, that the analogue of theorem 4.1.2 holds for Quest). The crucial point is the validity of the subsumption rule:

$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

This rule is valid simply because the interpretation of the term a, say, comes with the meaning of the entire judgment E ⊢ a : A or E ⊢ a : B. We

gave this meaning in such a way that it automatically *coerces* a to B in the semantics when interpreting E ∫ a : B. Indeed, the meaning of E ∫ a : A is an equivalence class in the p.e.r. ⟦E ∫ A type⟧'e (together with the assertion that it actually belongs to the class), while the meaning of ⟦E ∫ a:B⟧'e is an element of the p.e.r. ⟦E ∫ B type⟧'e, which is in general a larger equivalence class.

It is worth noticing the essential role of the interpretation of polymorphic types as intersections. The isomorphism between product and intersection in 5.1.1 is the core of this interpretation. (See the last two cases in 5.2.1.) It says that type erasing does not affect the meaning of polymorphic terms, modulo equivalence classes, and reduces the entire challenging business of how to apply a term to a type, to a simple type coercion in the model. That is, $\ulcorner n \urcorner_S \dagger A = \ulcorner n \urcorner_{G(A)}$, which interprets the polymorphic application for $S = \bigcap_{A \in K} G(A)$ (see 5.1.2), corresponds to coercing $\ulcorner n \urcorner_S$ to the generally larger equivalence class $\ulcorner n \urcorner_{G(A)}$.

This has a clear mathematical and computational meaning. Mathematically, it derives from the fact that the maps from any D-set with the full realizability relation to a p.e.r. are constant functions. (See [Longo Moggi 88], or prove it for exercise.) This is a simple feature inherited from a deep fact: the validity of the Uniformity Principle in the Realizability Universe, which is the categorical background of this construction [Longo 88]. Computationally, it says that at run time we disregard types, or that computations are type-free, in particular the computation of a polymorphic term. However, given a computation n of type $\bigcap_{A \in K} G(A)$, it happens that n is equivalent to more computations when updated to type A: namely, all those in $\ulcorner n \urcorner_{G(A)}$.

In [Bruce Longo 89] yet another interpretation of Fun, the progenitor of Quest, is given. The idea, in that paper, is to use the interpretation of the language with coercions in order to give meaning to the one without coercions. This is based on a series of theorems which relate *abbreviated* terms (that is, terms where all coercions are erased) to their *fattenings* (that is, terms where coercions are put back in place). More precisely, in our language, given $E \int_C a : A$, a judgment in $Quest_C$, *abbrev*(a) is obtained by erasing all coercions. Then, for E ∫ b : B in Quest, b' is a fattening when *abbrev*(b') = b. The BL-interpretation of the judgment E ∫ a : A in Quest, is given by setting:

$$BL⟦E \int a : A⟧e = ⟦E \int_C a' : A⟧e$$

where $⟦E \int_C a' : A⟧e$ is the semantics in part 4, for a fattening a' of a.

With some work, [Bruce Longo 89] showed that this is well defined. Indeed,

it coincides with our current interpretation [-]'. In other words, by the results in [Bruce Longo 89] and some further work, we claim that, given a model of the type-free l-calculus and the realizability structures over it as models of Quest, one has:

$$BL[E \int a : A]e = [E \int a : A]'e$$

Observe finally that this interpretation is "coherent", in the sense of [Curien Ghelli 89], since by definition it depends only on the proved judgment and not its derivation. More generally, the model satisfies the conditions in the *coherence theorem* in [Curien Ghelli 89].

6. Conclusions

We have described a formal system, which can be considered the kernel of the Quest language, and we have investigated a particularly attractive approach to its semantics. The formal system requires a lot of semantics models, probably more than any previous typed system. Fortunately, PER models promise to satisfy all the required features, and more (e.g. dependent types). More work needs to be done both on the syntactic side, studying the properties and the degree of completeness of the formal system, and on the semantic side, mostly with respect to recursion and recursive types.

Acknowledgements

We would like to thank Roberto Amadio and Kim Bruce. Working jointly and in parallel with them has provided us with a permanent source of ideas and inspiration. The many discussions with John Mitchell and P.-L. Curien have been essential for this work. We also thank Martín Abadi, Simone Martini, and Andre Scedrov for important suggestions, and Narciso Marti-Oliet for careful technical proofreading. Aspects of the formal system have been inspired by, and are still under investigation by many of the authors above.

References

[Abadi Plotkin 90] M.Abadi, G.D.Plotkin: **A Per model of polymorphism and recursive types**, LICS '90.

[Amadio 89] R.Amadio: **Recursion over realizability structures**, Information and Computation, to appear.

[Amadio 89a] R.Amadio: **Formal theories of inheritance for typed functional languages**, Note interne TR 28/89, Dipartimento di Informatica, Università di Pisa.

[Asperti Longo 91] A.Asperti, G.Longo: **Categories, types and structures:** an introduction to category theory for the working computer scientist, **M.I.T. Press,** to appear.

[Asperti Martini 89] A.Asperti, S.Martini: **Categorical models of polymorphism**, Note interne, Dipartimento di Informatica, Università di Pisa.

[Bainbridge Freyd Scedrov Scott 87] E.S.Bainbridge, P.J.Freyd, A.Scedrov, P.J.Scott: **Functorial polymorphism, preliminary report**, Proc. of the Programming Institute on Logical Foundations of Functional Programming, Austin, Texas, June 1987, and **T.C.S.** vol. 70, pp. 35-64, 1990.

[Barendregt 84] H. Barendregt: **The lambda calculus; its syntax and semantics**, Revised and expanded edition, North Holland.

[Breazu-Tannen Coquand Gunter Scedrov 89] V.Breazu-Tannen, T.Coquand, C.Gunter, A.Scedrov: **Inheritance and explicit coercion,** Proc. of the Fourth Annual Symposium on Logic in Computer Science, 1989.

[Bruce Longo 89] K.Bruce, G.Longo: **Modest models of Records, Inheritance and Bounded Quantification**, Information and Computation, to appear. (Preliminary version: CMU Report CS-88-126, Proceedings of LICS '88, Edinburgh, pp. 38-50).

[Carboni Freyd Scedrov 87] A.Carboni, P.J.Freyd, A.Scedrov: **A categorical approach to realizability and polymorphic types**, Proc. of the Third Symposium on Mathematical Foundations of Programming Language Semantics, New Orleans, to appear.

[Cardelli 88] L.Cardelli: **A semantics of multiple inheritance**, in Information and Computation 76, pp 138-164, 1988.

[Cardelli 89] L.Cardelli: **Typeful programming**, Lecture Notes for the IFIP Advanced Seminar on Formal Methods in Programming Language Semantics, Rio de Janeiro, Brazil, 1989. SRC Report #45, Digital Equipment Corporation, 1989.

[Cardelli Donahue Glassman Jordan Kalsow Nelson 88] L.Cardelli, J.Donahue, L.Glassman, M.Jordan, B.Kalsow, G.Nelson: **Modula-3 report**, Research Report n.31, DEC Systems Research Center, Sep. 1988.

[Cardelli Mitchell 89] L.Cardelli, J.C.Mitchell: **Operations on records**, Proc. of the Fifth Conference on Mathematical Foundations of Programming Language Semantics, New Orleans, 1989, to apppear.

[Cardelli Wegner 85] L.Cardelli, P.Wegner: **On understanding types, data abstraction and polymorphism**, Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.

[Cook Hill Canning 90] W.Cook, W.Hill, P.Canning: **Inheritance is not subtyping**, Proceedings of POPL'90, San Francisco.

[Curien Ghelli 90] P.L. Curien, G. Ghelli: **Coherence of subsumption**, to appear.

[Ehrhard 88] T.Ehrhard: **A Categorical Semantics of Constructions** Proceedings of LICS'88, Edinburgh.

[Fairbairn 89] J. Fairbairn: **Some types with inclusion properties in , ,** , Technical Report No. 171, University of Cambridge, Computer Laboratory.

[Feferman 87] S. Feferman: **Weyl Vindicated: Das Kontinuum, 70 Years Later**, preprint, Stanford University (Proceedings of the Cesena Conference on Logic and Philosophy of Science, to appear).

[Feferman 88] S.Feferman: **Polymorphic typed lambda-calculi in a type-free axiomatic framework**, Dept. of Mathematics, Journal of the ACM 151, 185, 30, 1 January.

[Freyd Mulry Rosolini Scott 90] P.J.Freyd, P.Mulry, G.Rosolini, D.Scott: **Domains in Per**, LICS '90.

[Girard 72] J-Y.Girard: **Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur**, Thèse de doctorat d'état, Université Paris VII, 1972.

[Hindley Longo 80] R. Hindley, G. Longo: **Lambda-calculus models and extensionality** Zeit. Math. Logik Grund. Math. n. 2, Vol. 26 (289-310).

[Hyland 82] M.Hyland: **The effective topos**, in The Brower Symposium, Troelstra and Van Dalen eds., North-Holland 1982.

[Hyland 87] M.Hyland: **A small complete category**, lecture delivered at the conference: Church's Thesis after 50 years, Zeiss (NL), June 1986. Annals of Pure and Applied Logic, 40, 1988.

[Hyland Pitts 87] J.M.E.Hyland, A.M.Pitts: **The theory of constructions: categorical semantics and topos-theoretic models**, in Categories in Computer Science and Logic (Proc. Boulder '87), Contemporary Math., Amer. Math. Soc., Providence RI.

[Longo 88] G.Longo: **Some aspects of impredicativity: notes on Weyl's philosophy of mathematics and today's Type Theory**, CMU Report CS-88-135, Lecture delivered at the Logic Colloquium 87, Ebbinghaus et al. eds, North Holland, Studies in Logic.

[Longo Moggi 88] G.Longo, E.Moggi: **Constructive Natural Deduction and its "-Set" interpretation**, CMU report CS-88-131, 1988.

[Luo 88] Z.Luo: **ECC, an Extended Calculus of Constructions**, Report, LFCS, Department of Computer Science University of Edinburgh.

[Martini 88] S.Martini: **Bounded quantifiers have interval models**, ACM Conference on Lisp and Functional Programming Languages, Snowbird, Utah.

[Meseguer 88] J.Meseguer: **Relating Models of Polymorphism**, SRI-CSL-88-13, October, SRI Projects 2316, 4415 and 6729, SRI International, Comp. sci. Lab.

[Mitchell 84] J.C.Mitchell: **Coercion and type inference**, Proc. POPL 1984.

[Mitchell 86] J. Mitchell:: **A type-inference approach to reduction properties and semantics of polymorphic expressions** ACM Conference on LISP and Functional Programming, Boston (308-319).

[Mitchell 88] J.Mitchell: **Polymorphic Type Inference and Containment**, Information and Computation, Vol. 76, Numbers 2/3, (211-249).

[Mitchell Plotkin 85] J.C.Mitchell, G.D.Plotkin: **Abstract types have existential type**, Proc. POPL 1985.

[Ohori 87] A.Ohori: **Orderings and types in databases**, Proc. of the Workshop on Database Programming Languages, Roscoff, France, September 1987.

[Pitts 87] A.Pitts: **Polymorphism is Set theoretic, constructively**, Symposium on Category Theory and Comp. Sci., SLNCS 283 (Pitts et al. eds.), Edinburgh.

[Reynolds 84] J C.Reynolds: **Polymorphism is not set-theoretic**, Symposium on Semantics of Data Types (Kahn, MacQueen, and Plotkin eds.) Lecture Notes in Computer Science 173, Springer-Verlag, 1984, pp. 145-156.

[Rosolini 86] G.Rosolini: **Continuity and effectiveness in Topoi**, D. Phil. Thesis, Oxford University.

[Scedrov 88] A.Scedrov: **A Guide to Polymorphic Types**, CIME Lectures Montecatini Terme, June, (revised version).

[Troelstra 73] A.Troelstra: Metamathematical investigation of Intuitionistic Arithmetic and Analysis. LNM 344, Springer-Verlag, Berlin.

[Wand 89] M.Wand: **Type inference for record concatenation and multiple inheritance**, Proc. of the Fourth Annual Symposium on Logic in Computer Science, 1989.