F-Bounded Polymorphism for Object-Oriented Programming

Peter Canning, William Cook, Walter Hill, Walter Olthoff
Hewlett-Packard Laboratories
P.O. Box 10490, Palo Alto, CA 94303-0971

John C. Mitchell
Department of Computer Science
Stanford University, Stanford, CA 94309

Abstract

Bounded quantification was introduced by Cardelli and Wegner as a means of typing functions that operate uniformly over all subtypes of a given type. They defined a simple "object" model and used bounded quantification to type-check functions that make sense on all objects having a specified set of "attributes." A more realistic presentation of object-oriented languages would allow objects that are elements of recursively-defined types. In this context, bounded quantification no longer serves its intended purpose. It is easy to find functions that makes sense on all objects having a specified set of methods, but which cannot be typed in the Cardelli-Wegner system. To provide a basis for typed polymorphic functions in object-oriented languages, we introduce *F-bounded quantification*. Some applications of F-bounded quantification are presented and semantic issues are discussed. Although our original motivation was to type polymorphic functions over objects, F-bounded quantification is a general form of polymorphism that seems useful whenever recursive type definitions and subtyping are used.

1 Introduction

Although object-oriented programming has attracted increasing interest in recent years, the development of polymorphic type systems for object-oriented languages has progressed slowly. One reason is that object-oriented languages are often described using terminology that sets them apart from functional languages. In addition, there has been a lack of formal models for object-oriented languages. As a result, it has been difficult to see how practical polymorphic type systems should be adapted for typing object-oriented constructs. In Cardelli's seminal paper [Car84], record subtyping was identified as an important form of polymorphism in object-oriented programs. This lead to the development of "bounded quantification" in [CW85]. If we view objects as elements of non-recursive record types, then bounded quantification provides a useful form of polymorphism over objects. However, a more sophisticated presentation of object-oriented constructs (as in [CCHO89, Coo89a, CDJ+89]) would allow objects that are elements of recursively-defined types. With recursive types, the Cardelli-Wegner form of bounded quantification is not sufficiently expressive to meet its original goal.

F-bounded quantification is a natural extension of bounded quantification that seems particularly useful in connection with recursive types. The essential idea may be illustrated by comparison with Cardelli-Wegner bounded quantification. Using "$\subseteq$" for the subtype relation, a simple example of a bounded-quantified type is the type $\forall t \subseteq \tau.\ t \to t$. This is the type of functions which map t to t, for every subtype t of τ. In a setting where all structurally similar objects belong to subtypes of a given type, many useful polymorphic functions will have bounded quantified types. For example, if we type an object by listing its methods and their types, an object with a print method may have type $\{\ldots, \mathsf{print{:}\,void} \to \mathsf{string}, \ldots\}$, indicating that the method print produces a print representation of the object. In the view of subtyping presented in [Car84], every type of this form will be a subtype of the type $\{\mathsf{print{:}\,void} \to \mathsf{string}\}$ of objects having only a print method. For example,

$$\{\mathsf{A{:}\,int} \to \mathsf{void},\ \mathsf{print{:}\,void} \to \mathsf{string}\} \subseteq \{\mathsf{print{:}\,void} \to \mathsf{string}\}.$$

A function taking a list of printable objects, a string and returning an object whose print representation matches the given string will have the bounded-quantified type $\forall t \subseteq \{\text{print: string}\}$. list[t] → string → t.

The flexibility of bounded quantification is dramatically reduced when objects belong to recursively-defined types. For example, consider the type

PartialOrder = { lesseq: PartialOrder → Bool }.

Each object of this type has a method lesseq which may be applied to another object of the same type. We intend the result of x.lesseq(y) to be true if x does not exceed y in some ordering, and false otherwise. Using lesseq, it should be possible to sort lists of PartialOrder elements, for example. We may easily write a sorting function with type $\forall t \subseteq$ PartialOrder. list[t] → list[t]. However, as we shall see later in the paper, object-oriented versions of the usual partially-ordered types such as integers are *not* subtypes of PartialOrder. Therefore, our generic sorting function cannot be used in the way we might expect. To solve this problem and related difficulties with other forms of recursive types we introduce a generalization of bounded quantification which we call *F-bounded quantification*, for want of better terminology. For the example at hand, consider the function F from types to types given by

F[t] = { lesseq: t → bool }.

We may write a polymorphic sorting function of type $\forall t \subseteq$ F[t].list[t] → list[t], where this function is defined on any type t such that $t \subseteq$ F[t]. Since integer $\subseteq$ F[integer], as explained in Section 3.3, we may apply such a sorting function to lists of integers or lists of any other type of objects having lesseq methods.

One practical application of F-bounded quantification is for understanding some problems in the type system of the Eiffel programming language [Mey88]. Eiffel introduces a special type expression, like Current, to represent recursion at the type level. Like Current always represents the 'current class': in a class P it simply stands for P, but when this class is inherited by a subclass C, like Current in an inherited attribute is reinterpreted to stand for the subclass C. The subclasses of such a class are usefully understood as having interfaces that satisfy an F-bound. However, as mentioned above, there is no necessary subtype relation among types satisfying an F-bound. This analysis explains some insecurities in the Eiffel type-system, which always allows subclasses to act as if they are subtypes of their parents. Further details are given in [Coo89b], along with some suggestions for correcting Eiffel's problems using the results in this paper. The error in the Eiffel type system illustrates the subtlety involved in designing flexible and sound type systems for object-oriented languages.

Another type system with a generalized form of bounded quantification is presented by Curtis [Cur87]. In this system, arbitrary collections of subtype constraints may be imposed on the quantified type variable. Thus Curtis' system subsumes our more modest proposal. However, we believe that our form of quantification is both sufficient to type-check practically useful object-oriented programs, and more tractable. Some evidence for the naturality of F-bounded quantification is the intriguing connection with F-coalgebras, the dual of a standard construction in the category-theoretic characterization of type recursion [SG82].

The next section surveys relevant background on strongly-typed object-oriented programming. Section 3 illustrates in detail problems with previous techniques for dealing with recursive object types. Section 4 introduces F-bounded quantification and demonstrates that it solves the typing problems presented in the previous section. some speculative observations on the semantic aspects of F-bounded quantification. Section 6 summarizes our contribution and indicates directions for future research.

2 Background

2.1 Objects, Records and Recursive Types

A fundamental premise underlying most models of object-oriented programming is that objects may be regarded as record whose components are functions representing methods [Car84, CW85, Wan88, Coo89a, CCHO89]. In this model, "message sending" is implemented by simple component selection.

Record types are used to describe the protocols [GR83] or interfaces of objects. A record type is a mapping of labels to types. A record consisting of labels $l_1, \ldots, l_j$ with associated values in types $\sigma_1, \ldots, \sigma_j$ has type $\{l_1{:}\sigma_1, \ldots, l_j{:}\sigma_j\}$. The fields of the record type describe messages together with the types of their arguments and return values. This view is adopted in the programming languages Amber [Car86], Modula-3 [CDJ+89], and TS [JGZ88].

Even the very simplest "textbook" examples from object-oriented programming produce recursively-defined record types [Rey75, BI82, Car86, JG88, CDJ+89]. For example, the type of a planar point with four methods is defined recursively as follows.

```
Point = {
    x : void → Real,
    y : void → Real,
    move : Real × Real → Point,
    equal : Point → Boolean
    }
```

The body of the recursive type is a record type indicated by braces {...}. The use of Point in the return type of move indicates that the move method returns an object that has the same type as the original point. Its use as the argument type of equal indicates that the equal method takes an argument with the same type. In other words, equal is a 'binary' operation on points, *viz.* for two points p and q, the expression p.equal(q) is meaningful.

A useful notation for recursively defined types is the form Rec $t.A$. Intuitively, the type Rec $t.A$ is the type t defined by $t = A$, where A generally contains the type variable t. Using this notation, we may define Point as follows.

```
Point = Rec pnt. {
    x : void  →  Real,
    y : void  →  Real,
    move : Real x Real  →  pnt,
    equal : pnt  →  Boolean
    }
```

Since the type variable pnt is bound by Rec, we may rename pnt without changing the meaning of this declaration.

2.2 Record Subtyping

Cardelli [Car84] identified record subtyping as an important form of polymorphism in object-oriented programming. One basic axiom of record subtyping may be written in the form

$$\{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k, \ldots, x_\ell{:}\sigma_\ell\} \subseteq \{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k\}$$

The main idea is that if a record r has fields $x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k$ and also $x_{k+1}{:}\sigma_{k+1}, \ldots, x_\ell{:}\sigma_\ell$, then in particular r has fields $x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k$. Therefore, any operation that makes sense on records of type $\{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k\}$ also makes sense on records of type $\{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k, \ldots, x_\ell{:}\sigma_\ell\}$. A generalization of this axiom is the inference rule

$$\frac{\sigma_1 \subseteq \rho_1, \ldots, \sigma_k \subseteq \rho_k,}{\{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k, \ldots, x_\ell{:}\sigma_\ell\} \subseteq \{x_1{:}\rho_1, \ldots, x_k{:}\rho_k\}}$$

which takes into account subtyping within the fields.

A standard rule is the function subtyping rule [Mit84, Car88]:

$$\frac{\sigma' \subseteq \sigma \qquad \tau \subseteq \tau'}{\sigma \to \tau \subseteq \sigma' \to \tau'}$$

Noting that the hypothesis $\sigma' \subseteq \sigma$ is opposite to the others, one says that the arrow constructor is *contravariant* in its first argument. One subtype rule for recursive types is the following [Car86]:

$$\frac{\Gamma, s \subseteq t \vdash \sigma \subseteq \tau}{\Gamma \vdash \mathsf{Rec}\ s.\ \sigma \subseteq \mathsf{Rec}\ t.\ \tau} \qquad \begin{array}{l} s \text{ free only in } \sigma. \\ t \text{ free only in } \tau. \end{array}$$

This means, informally, that if assuming s is a subtype of t allows one to prove that σ is a subtype of τ, then the recursive type Rec $s.\ \sigma$ is a subtype of Rec $t.\ \tau$. For example, the following type is a supertype of Point:

```
Movable = Rec mv. { move : Real x Real  →  mv }
```

because the body of Point is a subtype of the body of Movable, given the assumption pnt ⊆ mv:

$$\frac{\begin{array}{l}\mathsf{pnt} \subseteq \mathsf{mv} \vdash \{\mathsf{move}{:}\,\mathsf{Real}\times\mathsf{Real} \to \mathsf{pnt}, \ldots\} \\ \qquad\qquad \subseteq \{\mathsf{move}{:}\,\mathsf{Real}\times\mathsf{Real} \to \mathsf{mv}\}\end{array}}{\begin{array}{l}\mathsf{Rec\ pnt.}\{\mathsf{move}{:}\,\mathsf{Real}\times\mathsf{Real} \to \mathsf{pnt}, \ldots\} \\ \qquad \subseteq \mathsf{Rec\ mv.}\{\mathsf{move}{:}\,\mathsf{Real}\times\mathsf{Real} \to \mathsf{mv}\}\end{array}}$$

2.3 Bounded Quantification

Cardelli and Wegner's language FUN [CW85] uses bounded quantification to type polymorphic functions over simple "objects" represented by records. Extensions to the language removed its reliance upon record field assignment, so that record operations may be expressed functionally [Wan87, JM88, Rem89]. The use of bounded quantification may be illustrated with a simple type of cartesian point objects:

```
SimplePoint = { x: int, y: int }
```

Note that simple points do not have any methods which take simple points as arguments or return simple points as results. Consequently, the type SimplePoint does not require a recursive type declaration.

A function that 'moves' simple points may be defined using bounded quantification. Move is a function of type ∀t ⊆ SimplePoint. t → Real x Real → t and is defined by the expression following the equals sign.

```
move : ∀t ⊆ SimplePoint. t → Real x Real → t
    = Fun[t ⊆ SimplePoint] fun(p:t) fun(dx,dy:Real)
        p with { x = p.x + dx, y = p.y + dy }
```

The notation Fun[t ⊆ SimplePoint] indicates that the first argument of move is required to be a subtype of SimplePoint. The second argument must be a value in this subtype. Its third argument is a pair of numbers representing the distance to be moved. The result of the function is computed by building a new record having the fields in the original subtype value, but with updated x and y components. This new record has the same type as the original argument to move.

Every subtype of SimplePoint is a legal argument to move. For example, values of type SimpleColoredPoint,

SimpleColoredPoint = { x: int, y: int, color: int }

are valid arguments to move, and the result of the application is also a SimpleColoredPoint.

3 Recursive Types and Bounded Quantification

3.1 Introduction

In this section we investigate the use of bounded quantification in polymorphic functions over objects with recursive types. We show that bounded quantification does not provide the same degree of flexibility in the presence of recursion as it does for non-recursive types.

Two kinds of problems are identified, depending upon the location of the recursion variable within the recursive type. In describing the two possibilities, it is useful to adopt the standard notion of "polarity" from logic. In a type expression $\sigma \to \tau$, the subexpression τ occurs *positively* and the subexpression σ *negatively.* If σ' occurs with positive or negative polarity in σ, then this occurrence will have the opposite polarity in $\sigma \to \tau$. A subexpression of τ will have the same polarity in $\sigma \to \tau$. For example, t is positive in $(t \to \sigma) \to \tau$ but negative in $t \to (\rho \to \tau)$. Polarity is preserved in record type expressions, so that t is positive and s is negative in $\{\mathsf{put}{:}t \to \mathsf{void},\ \mathsf{get}{:}\mathsf{void} \to s\}$.

3.2 Subtyping and Positive Recursion

When the recursion variable of a recursive type appears positively, subtyping does not ensure the intuitively expected typing behavior. Consider the recursive type Movable introduced above. The recursion variable mv in Movable only occurs positively.

Movable = Rec mv. { move: Real × Real → mv }

It is reasonable to define a function, translate, that moves a movable value one unit in both directions:

translate = fun(x:Movable) x.move(1.0, 1.0)

Although this function works for any value whose type is a subtype of Movable, the result of the function application is always of type Movable, according to the typing rules of [CW85]. It would be preferable to have a polymorphic translate which, for any subtype of t of Movable, takes argument of type t and return a value of the same type. However, an easy semantic argument shows that translate as defined above does *not* have the type

translate : ∀r ⊆ Movable. r → r

To see this, consider the type

R = { move: Real × Real → Movable, other: A }

It is easy to see that R is a proper subtype of Movable. However, if we apply translate to an object of type R, we obtain an object of type Movable, not R. Thus the best we can say with bounded quantification is

translate : ∀r ⊆ Movable. r → Movable

which is no more general than the ordinary function type Movable → Movable.

The careful reader may notice that translate can in fact be typed without using bounded quantification, giving

translate : ∀t. { move: Real × Real → t } → t

However, this should not be regarded as a defect in our presentation; this works only because translate is an unusually simple example. An essential aspect of translate is that the parameter x only occurs once in the body of the function, where we access the move field. In a more complicated function like

```
choose = fun(b:bool) fun(x:Movable)
    if b then x.move(1.0, 1.0) else x
```

in which the method is called and the object returned, the simple typing without bounded quantification is not possible.

3.3 Subtyping and Negative Recursion

For a recursive type with a negative recursion-variable, the intuitive concept of 'adding fields' to produce subtypes does not work: the resulting types are not subtypes of the original recursive type. Consequently, bounded quantification cannot be used to quantify over these types. To illustrate, assume we want to define a polymorphic minimum function on a PartialOrder type that describes values with a comparison method:

PartialOrder = Rec po. { lesseq: po → bool }

minimum : ∀t ⊆ PartialOrder. t → t → t

The minimum function should return the lesser of its two arguments, determined by asking the first argument to compare itself with the second. Intuitively, values of type Number or String should be admissible arguments for the polymorphic minimum, since they both have a lesseq operation as required. The type Number, in our view of object-oriented languages, is a recursively defined record type:

Number = Rec num. { ..., lesseq: num → bool, ...}

However, the polymorphic application minimum [Number] is type-incorrect, because Number is not a subtype of PartialOrder. If we try to derive Number $\subseteq$ PartialOrder by unrolling the two types we obtain

$$\{\ \ldots,\ \text{lesseq: Number} \rightarrow \text{bool}, \ldots\} \subseteq \{\ \text{lesseq: PartialOrder} \rightarrow \text{bool}\ \}$$

which requires PartialOrder $\subseteq$ Number. This is contrary to what we wanted to show, indicating that Number $\subseteq$ PartialOrder is not derivable unless Number = PartialOrder.

One type that is a subtype of PartialOrder is

$$\text{Rec t.}\ \{\ \ldots,\ \text{lesseq: PartialOrder} \rightarrow \text{bool}, \ldots\}$$

An object of this type could be compared (using lesseq) with any other value of type PartialOrder, but since PartialOrder does not provide any fields on which to base this comparison, objects of this type have little practical value. In situations where more fields are present such types may be useful, but the problem remains that subtyping cannot capture the intuitive polymorphism desired for minimum.

4 F-bounded Quantification

4.1 Introduction

F-bounded quantification allows the practical examples given above to be type-checked with intuitively desirable types. We say that a universally quantified type is F-bounded if it has the form

$$\forall t \subseteq F[t].\sigma$$

where $F[t]$ is an expression, generally containing the type variable t. The semantics of F-bounded quantification are discussed briefly in Section 5.

F-bounded polymorphic types differ from ordinary bounded types by binding the type variable in both the result-type σ and the type bound $F[t]$. If $F[t]$ is a type of the form $F[t] = \{a_i{:}\,\sigma_i[t]\}$, then the condition $A \subseteq F[A]$ says, in effect, that A must have the methods a_i and these methods must have arguments as specified by $\sigma_i[A]$, which are defined in terms of A. Thus A will often be a recursive type, suggesting that F-bounded quantification is closely related to type recursion. But bounded quantification $\forall t \subseteq (\text{Rec } r.F[r]).\sigma(t)$ over a recursive type is very different from the F-bounded quantification $\forall t \subseteq F[t].\sigma(t)$ over the type-function F that defines the recursive type, as shown in the following sections.

4.2 Positive Recursion

As we saw in Section 3.2, the polymorphic application translate[Point] produces a function of type Point $\rightarrow$ Movable, rather than Point $\rightarrow$ Point as desired. A simple type derivation will both motivate the definition of F-bounded quantification, and show how it can be used to achieve the desired typing of translate.

In the this example we 'work backwards' to derive the F-bounded constraint from the typing problem posed by translate. The problem is to derive a condition on a type t so that for any variable x of type t, x.move(1.0, 1.0) has type t. In the following discussion we use the subtype rules of [Car88] or [Mit84]. We are looking for the minimal condition on t such that the following typing can be derived:

$$\text{x} : \text{t} \vdash \text{x.move(1.0, 1.0)} : \text{t}$$

By the application (APP) and selection (SEL) rules, this reduces to

$$\text{x} : \text{t} \vdash \text{x} : \{\ \text{move} : \text{Real} \times \text{Real} \rightarrow \text{t}\ \}$$

Using the subtyping rule we then derive

$$\frac{\tau \subseteq \{\text{move: Real} \times \text{Real} \rightarrow \text{t}\}}{\text{x:t} \vdash \text{x:}\tau}$$

Since the type τ does not occur in any other assumption, we may simplify to the requirement

$$\text{t} \subseteq \{\ \text{move} : \text{Real} \times \text{Real} \rightarrow \text{t}\ \},$$

which cannot be proved without additional assumptions. Expressing this condition as t $\subseteq$ F-Movable[t], where

$$\text{F-Movable[t]} = \{\ \text{move} : \text{Real} \times \text{Real} \rightarrow \text{t}\ \}$$

it is clear that this condition fits the format of F-bounded quantification. Motivated by the preceding discussion, we define the F-bounded polymorphic function:

```
translate = Fun[ t ⊆ F-Movable[t] ]
              fun(x:t) x.move(1.0, 1.0)
```

with F-bounded polymorphic type

$$\text{translate} : \forall \text{t} \subseteq \text{F-Movable[t]}\ .\ \text{t} \rightarrow \text{t}$$

Since Point $\subseteq$ F-Movable[Point], the application translate[Point] is type-correct, and has result of type Point $\rightarrow$ Point. Of course translate will also work for other types that satisfy the constraint t $\subseteq$ F-Movable[t], such as ColoredPoint, defined as follows.

```
ColoredPoint = Rec pnt. {
    x : void → Real,
    y : void → Real,
    c : void → Color,
    move : Real × Real → pnt,
    }
```

It is interesting to note that the type function F-Movable is related to the recursive type

```
Movable = Rec mv. { move: Real × Real → mv }
```

that failed to provide the desired typing in Section 3.3. F-Movable is constructed syntactically by regarding the body of the recursive type expression as a type function.

4.3 Negative Recursion

In Section 3.3, we saw that Number is not a subtype of PartialOrder. Nevertheless, the types Number and String, as well as the type PartialOrder all have binary operations lesseq. As a consequence, the expression x.lesseq(y) is well-typed if x and y both have one of these types, but not if x and y have different types. However, using ordinary bounded quantification, it does not seem possible to define a polymorphic minimum function that works correctly for the types PartialOrder, Number or String. In this section, we will see that F-bounded quantification allows us to define such a polymorphic minimum function. This is an important advance, since current typed object-oriented languages are notoriously restrictive when it comes to binary operations.

The common structure among PartialOrder, Number and String may be described using a type function derived from the recursive definition of PartialOrder:

```
F-PartialOrder[t] = { lesseq: t → bool }.
```

Applying F-PartialOrder to Number we see that

```
F-PartialOrder[Number] = { lesseq: Number → bool }
```

and hence

```
Number ⊆ F-PartialOrder[Number]
```

Although Number is not a subtype of PartialOrder, it is a subtype of F-PartialOrder[Number], which is all we need to compute minimum. Forming the F-bounded polymorphic function

```
minimum = Fun[t ⊆ F-PartialOrder[t]]
        fun(x:t) fun(y:t) if x.lesseq(y) then x else y
```

with type

```
minimum : ∀t ⊆ F-PartialOrder[t] . t → t → t
```

we capture a form of polymorphism which does not seem possible with ordinary bounded quantification.

Although we have discussed negative and positive recursion separately, F-bounded quantification also works when both occur in the same recursive type. While the following statement is technically imprecise, it seems intuitively helpful to say that F-bounded quantification characterizes the types that have "recursive structure" similar to the type Rec $t.F[t]$. Intuitively, a type $F[A]$ describes a set of meaningful operations, possibly accepting values of type A as arguments or returning such values as results. Elements of type A have these operations if we may view each element of A as an element of $F[A]$, *i.e.*, $A \subseteq F[A]$.

One type that always satisfies $A \subseteq F[A]$ is the recursive type $A = \text{Rec } t.F[t]$. More generally, if $G[t]$ is a type expression and $G[t] \subseteq F[t]$ for all t, then the recursive type $A = \text{Rec } t.G[t]$ also satisfies $A \subseteq F[A]$. This follows from the observation that if $G[t] \subseteq F[t]$ for all t, then $A = G[A] \subseteq F[A]$. However, it is worth noting that (depending on F) there may be other types satisfying $t \subseteq F[t]$ which do not have this form.

5 Semantics

There several ways of developing semantics for F-bounded quantification. We have not explored any of these in detail. Here we will simply sketch some approaches to semantics and an intriguing connection to F-algebras.

For a direct semantics, it is useful to have a way to denote the family of all types that satisfy the bound $t \subseteq F[t]$ for each F. This collection of types would constitute a ***kind***, in the sense of [BL88, BMM89], analogous to the POWER kind of [Car88]. This may be achieved by defining a kind constructor FBOUND : (TYPE → TYPE) → KIND with intuitive interpretation

$$\text{FBOUND}[F] = \{t \mid t \subseteq F[t]\}$$

Given this constructor, the type $\forall t \subseteq F[t].\sigma$ may be interpreted using kinded quantification as $\forall t{:}\text{FBOUND}[F].\sigma$. We see no problem in incorporating this into the model definition of [BMM89]. Another view may be derived from constrained quantification of Curtis [Cur87], since F-bounded quantification is subsumed by his system.

An alternative approach is to use the semantics-by-translation of Breazu-Tannen *et.al.* [BCGS89]. In their semantics, bounded type-derivations in the language with bounded quantification are translated into type-derivations in a simpler calculus with explicit coercions. In particular, the type $\forall t \subseteq \tau.\ \sigma$ is translated to $\forall t.\ (t \to \tau) \to \sigma$, in which $t \to \tau$ is the explicit coercion.

Since τ is in the scope of the universal quantification of t, there is no problem with allowing τ to have the form $F[t]$ in the translated language. One technical point in [BCGS89] is a coherence condition requiring, intuitively, that any two type-derivations for an F-bounded term must translate into provably equivalent terms in the calculus with explicit coercions. Coherence is a difficult technical property, and we have not verified it for our calculus.

Regardless of how we interpret a subtyping assertion $A \subseteq B$, it is clear that this assertion implies some kind of map from A to B. This simple observation leads us to some interesting connections between F-bounded quantification and the standard category-theoretic machinery associated with recursive type definitions [SG82]. To begin with, in most semantics of recursive types, it is possible to extend type functions defined by type expressions to functors (maps on types and functions) over some category (perhaps with a more limited choice of functions than we actually define in programming). If we have a functor F and wish to find a type t satisfying $t = F[t]$, where $=$ means isomorphism, then it suffices to find an *initial F-algebra,* where an F-algebra is a pair $\langle t, f\rangle$ with $f: F[t] \to t$. It is an easy exercise to prove that if $\langle t, f\rangle$ is an initial F-algebra, then f has a two-sided inverse f^{-1}. The dual of an F-algebra is an F-coalgebra, which is a pair $\langle t, f\rangle$ with $f: t \to F[t]$. The argument showing that an initial F-algebra is a solution to $t = F[t]$ also shows that the final F-coalgebra satisfies $t = F[t]$.

In F-bounded polymorphism, we quantify over all types t with $t \subseteq F[t]$. Taking into account that $t \subseteq F[t]$ implies some kind of map from t to $F[t]$, this means we are essentially quantifying over pairs $\langle t, f\rangle$ with $f: t \to F[t]$, or some family of F-coalgebras. (The quantification over some family of maps $t \to F[t]$ is made explicit in the [BCGS89] translation.) Since the recursive type Rec t. $F[t]$ may be regarded as a particular F-coalgebra, this suggests that F-bounded polymorphism involves quantification over a category whose objects are properly regarded as "generalizations" of the recursive type Rec t. $F[t]$.

One way of seeing why this provides useful polymorphism in object-oriented languages is to consider the typing rules associated with recursive types. If $t =$ Rec t. $F[t]$, then we have an "introduction" rule saying that if an expression $e: F[t]$ then $e: t$. The "elimination" rule gives the converse: if $e: t$ then $e: F[t]$. These rules are based on the two directions of the isomorphism $t = F[t]$. If A satisfies the F-bounded condition $A \subseteq F[A]$, then A is a type which has the "elimination" typing rule associated with Rec t. $F[t]$, but not necessarily the associated "introduction" rule. This is a precise way of saying that at type A satisfying $A \subseteq F[A]$ shares "structural similarity" with the recursive type Rec t. $F[t]$. In general, for recursive types of the form

$$\mathsf{Rec}\ t.\{\mathsf{method}_1: \sigma_1, \ldots, \mathsf{method}_k: \sigma_k\}$$

it seems that only the "elimination" is needed to make meaningful use of object with this type. Hence F-bounded quantification seems to be "exactly what we need" in order to type polymorphic functions over objects with recursive types.

6 Conclusion

We have identified a generalization of bounded quantification, called F-bounded quantification, in which the bound type variable may occur within the bound. We argue that F-bounded quantification is useful for typing programs involving recursive types: it allows quantification over types that are "structurally similar" to the recursive type Rec t. $F[t]$.

As directions for future work, we note that F-bounded quantification is closely related to inclusion for single-sorted algebraic signatures. F-bounded quantification captures the notion of adding more operations to a recursive type while preserving the recursive structure of the type.

F-bounded quantification also has an impact on the relation between inheritance and subtyping in object-oriented programs. As noted in Section 4.3, two types t_1 and t_2 may satisfy an F-bound ($t_1 \subseteq F[t_1]$ and $t_2 \subseteq F[t_2]$) but not be in a subtype relation (neither $t_1 \subseteq t_2$ or $t_2 \subseteq t_1$). This means that a F-bounded function may be applied to (or "inherited" by) objects with incomparable types, demonstrating that the inheritance hierarchy is distinct from the subtype hierarchy [Sny86]. In the Abel project at HP Labs, we are exploring the consequences of this separation on programming language design.

References

[BCGS89] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 112–133, 1989.

[BI82] Alan H. Borning and Dan H. Ingalls. A type declaration and inference system for Smalltalk. In *Proc. of Conf. on Principles of Programming Languages*, pages 133–141, 1982.

[BL88] Kim Bruce and G. Longo. A modest model of records, inheritance and bounded quanti-

fication. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 38–50, 1988.

[BMM89] Kim B. Bruce, Albert R. Meyer, and John C. Michell. The semantics of second-order lambda calculus. *Information and Computation*, (to appear).

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, LNCS 173*, pages 51–68. Springer-Verlag, 1984.

[Car86] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages, LNCS 242*, pages 21–47, 1986.

[Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *Conf. Rec. ACM Symp. on Pinciples of Programming Languages*, pages 70–79, 1988.

[CCHO89] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, 1989 (to appear).

[CDJ+89] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kaslow, and Greg Nelson. The Modula-3 type system. In *Conf. Rec. ACM Symp. on Pinciples of Programming Languages*, pages 202–212, 1989.

[Coo89a] William Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, March 1989.

[Coo89b] William Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, 1989 (to appear).

[Cur87] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis (Draft)*. PhD thesis, Cornell, 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[GR83] Adele Goldberg and Dave Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[JG88] Ralph Johnson and Justin Graver. A user's guide to Typed Smalltalk. Technical Report UIUCDCS-R-88-1457, University of Illinois, 1988.

[JGZ88] Ralph Johnson, Justin Graver, and L. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, 1988.

[JM88] L. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM conf. on Lisp and Functional Programming*, 1988.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[Mit84] John C. Mitchell. Coercion and type inference (summary). In *Conf. Rec. ACM Symp. on Pinciples of Programming Languages*, 1984.

[Rem89] Dedier Remy. Typechecking records and variants in a natural extension of ML. In *Conf. Rec. ACM Symp. on Pinciples of Programming Languages*, pages 77–88, 1989.

[Rey75] John Reynolds. User-defined data types and procedural data structures as complimentary appraches to data abstraction. In *New Advances in Algorithmic Languages*. IRIA, 1975.

[SG82] M. B. Smyth and G. D. Gordon. The category-theoretic solutions of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 38–45, 1986.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[Wan88] Mitchell Wand. Type inference for objects with instance variables and inheritance, 1988. manuscript.