STUDIA UNIV. "BABEŞ–BOLYAI", INFORMATICA, Volume **XLIV**, Number 2, 1999

F-BOUNDED QUANTIFICATION AND THE MATCHING RELATION

SIMONA MOTOGNA

Abstract. The introduction of F-bounded polymorphism had proved to have great benefits in specification of object oriented languages, and the most important one, concerning binary methods and recursion definition, is also presented here. Then many other systems had appeared, such as PolyTOIL, based on the matching relation, and we prove that F-bounded quantification and the matching relation are equivalent.

1. F-bounded polymorphism

The extension proposed by the Abel group [CHC90] is based on introducing recursive types. The specification of polymorphic functions over objects cannot be done correctly through bounded quantification, and the recursive functions give a suitable solution. That's why we will study their behaviour in the next paragraph.

1.1. **Subtyping and recursion using bounded quantification.** The bounded quantification was introduced in the language Fun [CW85] in order to type polymorphic functions, that were defined over "simple" objects, represented as records. But in most cases, class definitions include the so called binary methods, namely methods with parameters of type representing that class. This situation imposes the condition that objects should be described as recursive records[CHC90], destroying their "simplicity". We will see what happens with the behaviour of the polymorphic functions over recursive objects and we will show that, in this case, bounded quantification fail to produce a correct answer [Ghel93].

When describing the recursive types, the two possible situations can be specified using the notion of polarity, that is "borrowed" from logic [CCHOM89].

In a type expression $s \to t$, the subexpression s occurs *negatively*, and the subexpression t occurs *positively*.

Let's consider a recursive type $SortList$, representing a sorted chained list, whose elements are of an arbitrary type t. We have also defined a method for inserting elements in the list, and the list is described as having a *head* (an element

1991 *CR Categories and Descriptors.* D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *Recursion.*

of type t) and the rest of the list, the *tail* of type $SortList$. In addition, in order to have a sorted list, it is necessary to have a total order relation defined between the list elements, so the type t will be a subtype of the type:

$$TotalOrder = \lambda\sigma.\{smaller : \sigma \to Bool, equal : \sigma \to Bool\}$$

(the method *insert* will be written knowing that each element is comparable with the others).

So, the sorted chained list will have the following form:

$$SortList = \forall t \leq TotalOrder.\mu sl.\{insert : t \to sl, head : t, tail : sl\}$$

If we want to have such a list of integer numbers, than the parameter t must be replaced with Int:

$$SortIntList = \mu sil.\{insert : Int \to sil, head : Int, tail : sil\}$$

that seems to be intuitively correct.

But, if t was replaced by Int, then the following condition must be satisfied:

$$Int \leq TotalOrder$$

and the type Int is, in general, defined as:

$$Int = \mu n.\{smaller : n \to Bool, equal : n \to Bool, ...\}$$

and if we decompose under the recursion variable, we obtain:

$$\begin{array}{ll} \{ & smaller : Int \to Bool, equal : Int \to Bool, ...\} \leq \\ & \{smaller : TotalOrder \to Bool, equal : TotalOrder \to Bool\} \end{array}$$

In order to satisfy this equation, each common component from the two records must obbey the subtyping rule for functions:

$$\frac{s \leq s', t' \leq t}{s' \to t' \leq s \to t}$$

and we obtain:

$$TotalOrder \leq Int$$

which represents a contradiction of what we really want.

So, we notice that the only valid substituion for t is $TotalOrder$. We can end with the following conclusion: bounded quantification does not specify correctly the behaviour of the polymorphic function defined over recursive objects.

1.2. **F-bounded quantification. Definition 1.1**: We say that a universal quantified type is ***F-bounded*** if it has the following form:

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

where $F[t]$ is an expression that, in general, contains the type variable t.

The F-bounded polymorphic types differ from the ordinary bounded types in the sense that both the result type σ and the bound $F[t]$ of the type depend upon the type variable t.

If $F[t]$ is a type of the form $F[t] = \{a_i : \sigma_i[t]\}$, then the condition $A \leq F[A]$ says, in fact, that A must contain all the methods a_i and these methods should have the arguments specified by $\sigma_i[A]$, that are defined depending on A. In conclusion, A will often be a recursive type, suggesting that the functional bounded quantification is strongly connected with type recursivity.

Intutively, F-bounded quantification characterizes the types that have a "recursive structure" similar to the type $\mu t.F[t]$. The type $F[A]$ describes a set of operations, which can accept values of type A as arguments and may return such values as results. The elements of type A have these operations, if we consider each element of type A as an element of type $F[A]$, namely $A \leq F[A]$.

A type that always satisfies $A \leq F[A]$ is the recursive type $A = \mu t.F[t]$. More generally, if $G[t]$ is a type expression and $G[t] \leq F[t]$ for any t, then the recursive type $A = \mu t.G[t]$ satisfies $A \leq F[A]$.

The F-bounded quantification has a major impact upon the relation between inheritance and subtyping in object oriented programming: two types t_1 and t_2 can satisfy a F-bound ($t_1 \leq F[t_1]$ and $t_2 \leq F[t_2]$) but may not be in a subtyping relation ($t_1 \not\leq t_2$ and $t_2 \not\leq t_1$). This means that a F-bounded function can be applied to (or inherited by) an object with incomparable types, proving that the inheritance hierarchy is different from the subtyping hierarchy.

The F-bounded polymorphism will allow, in general, to write functions that work in the same time on objects belonging to classes that are in an inheritance relation, in the same way in which bounded polymorphism allows to write functions that work in the same time on types and their subtypes.

1.3. **Subtyping and recursion using F-bounded polymorphism.** In the first paragraph we noticed that Int is not a subtype of the type $TotalOrder$. However, the types Int and $TotalOrder$ have the same binary operations: $smaller$ and $equal$. So, the expression $x.smaller(y)$ is correctly typed if x and y have both one of these two types and incorrectly typed if they have different types.

In the following, we will see that for such a behaviour the subtyping relation between the two types is not necessary, and we can introduce another relation that guarantees the desired behaviour.

This common structure of the two types can be described through a functional type, derived from the recursive definition of the type $TotalOrder$. This functional

type represents a F-bound. The "F-" notation before the type indicates that it represents a F-bound:

$$F - TotalOrder[t] = \{smaller : t \to Bool, equal : t \to Bool\}$$

Applying this function for Int we obtain:

$$F - TotalOrder[Int] = \{smaller : Int \to Bool, equal : Int \to Bool\}$$

and

$$Int \leq F - TotalOrder[Int]$$

because the contravariance is respected for each component: $Int \leq Int$.

We can now build the F-bounded polymorphic definition of the sorted chained list:

$$\begin{aligned} SortList \quad = \quad & \forall t \leq F - TotalOrder[t].\mu sl. \\ & \{insert : t \to sl, head : t, tail : sl\} \end{aligned}$$

and then

$$SortIntList = \mu sil.\{insert : Int \to sil, head : Int, tail : sil\}$$

will be a valid definition.

Let's notice that $Int \not\leq TotalOrder$, but $Int \leq F - TotalOrder[Int]$ which is enough.

As a conclusion we may say that F-bounded quantification allows us to use generic polymorphism together with inheritance in object-oriented languages: we have a generic class $SortList$ that can be specialised substituting the parameter, obtaining, for example, $SortIntList$. Also, the other operation for creating derived classes, adding new characteristics, will function correctly. For example, if we want to obtain a merged sorted list, we have:

$$\begin{aligned} MergedSortList \quad = \quad & \forall t \leq F - TotalOrder[t].\mu msl. \\ & \{insert : t \to msl, head : t, tail : msl, merge : msl \to msl\} \end{aligned}$$

The recursion variable will ensure that the methods always return the desired type, namely $MergedSortList$ and any object of this class is a member of class $SortList$.

2. The matching relation

Kim Bruce's contribution to object oriented programming specification consists in the description of the type systems and the semantics of several object oriented languages, each of them introducing new elements: TOOPLE [Bruc93], TOIL [BvG93] and the most remarkable one PolyTOIL [BSvG95].

PolyTOIL is an object oriented programming language, polymorphic, with static typing, and its type system had been proved to be correct.

Bruce's idea that inovates this domain is the separation of the subtyping from the inheritance definition, assigning name to the type of $self$ and defining the type checking rules. Inheritance is no longer expressed as subtyping (the system contains two hierarhies, one for subtypes and one for subclasses) but as a *matching* between the object types and is defined as follows:

Definition 2.1 $ObjectType\ \tau'$ *matches* (is in a *matching* relation) with $ObjectType\ \tau$, denoted $ObjectType\ \tau' <\#\ ObjectType\ \tau$, if for each method name m from τ there exists a corresponding method in τ' and m's type in τ' is a subtype of the method type from τ;

or (more formally)

$$ObjectType\,\{m_j : S_j\}_{1\le j\le m} <\#\ ObjectType\,\{m_i : T_i\}_{1\le i\le n}$$

if and only if

$$n \le m \text{ and for every } i \le i \le n, S_i \le T_i$$

where $ObjectType\,\{m_j : S_j\}$ represents the type of objects that have methods m_j with type S_j.

We will notice that the definition doesn't contain instance variables, since it is assumed that they are not visible from outside and can be accessed only through the object's method.

The subtyping relation is denoted as $S \le T$. If SC is a subclass of the class C then $SCType <\# CType$.

The subtyping relation is defined as follows [BSvG95]:

Definition 2.2 $ObjectType\,\{m_j : S_j\}_{1\le j\le m} \le ObjectType\,\{m_i : T_i\}_{1\le i\le n}$ if and only if

1. $n \le m$ and for every $1 \le i \le n, S_i \le T_i$
2. $\forall T_i, 1 \le i \le n, T_i$ is NOT contravariant to $MyType$ (there are no parameters of type $MyType$, the type of the $self$ variable, in supertypes).

Remark 2.1: The only difference between subtyping and matching is the second condition from the definition. So, if two object types are in a subtyping relation then they are in a matching relation, but the reverse statement is false [AC95].

3. The equivalence between the matching relation and F-bounded quantification

Theorem 2.1 [Moto00] If there exists a matching relation between two object types:

$$ObjectType\ t' <\#\ ObjectType\ t$$

then *ObjectType* t is a F-bound for *ObjectType* t' and the reverse statement is also true, namely there exists a matching relation between an object type and its F-bound.

Proof:
"$\Rightarrow$"

Let *ObjectType* t be the object type with the following form: $\{m_i : T_i\}_{1 \leq i \leq n}$. For every $1 \leq i \leq n$, the type T_i has the following form: $\sigma_i \rightarrow \tau_i$, so they are functional types, because m_i represent methods (*ObjectType* t is an object type, and the instance variables are hidden and only the methods are visible). In addition, σ_i and/or τ_i can depend on $MyType$, which represents the $self$ type of the object. The $self$ variable can be considered as a recursion variable and then we can describe the object type as follows:

$$ObjectType\ t = \mu MyType.\{m_i : T_i[MyType]\}_{1 \leq i \leq n}$$

If we create an object of type *ObjectType* t' of the form: $\{m_j : S_j\}_{1 \leq j \leq m}$, that inherits from *ObjectType* t, then the two types are in matching relation and:

$$n \leq m \text{ and } S_i \leq T_i.$$

On the other hand, if we use F-bounded quantification, any object definition based on the object type *ObjectType* t must respect the F-bound condition:

$$ObjectType\ t' \leq F - ObjectType\ t[t'] \tag{1}$$

Using the model described in the paragraph 1.3, the computation of the F-bound will produce the following result:

$$F - ObjectType\ t[MyType] = \{m_i : T_i\}_{1 \leq i \leq n}$$

and the condition (1) is satisfied because *ObjectType* t' has all the m_i methods (and probably more), and the arguments of these methods are correctly specified, since $S_i \leq T_i$.

In conclusion, the proof of this implication consists in noticing that $MyType$ implies a recursive definition of the type.

"$\Leftarrow$"

According to the definition of F-bounded quantification, we have: if $F[t]$ is a type of the form $F[t] = \{a_i : T_i[t]\}_{1 \leq i \leq n}$, then the condition $A \leq F[A]$ says, in fact, that: [1] A must contain all a_i methods and [2] these methods must have the arguments specified by $T_i[A]$, which are defined depending on A.

So, from the condition [1] we obtain that *ObjectType* t' will have the form $\{a_i : S_i, a_{n+1} : S_{n+1}, ...\}$, and from [2] we can deduce that the types S_i, for $1 \leq i \leq n$, are obtained substituting the t parameter with the object type, so:

$$S_i = T_i[t/ObjectType\ t']$$

and it is obvious that $S_i \leq T_i$.

So, if

$$ObjectType\ t' \leq F - ObjectType\ t[ObjectType\ t']$$

then

$$ObjectType\ t' < \#\ F - ObjectType\ t[ObjectType\ t']$$

Remark 2.2: This implication can be proved easier if we use ***Remark 2.1***, that states that if two object types are in a subtyping relation, then they are also in a matching relation and noticing that a F-bound respects the structure of an object type.

References

[AC95] M Abadi, L Cardelli - *On subtyping and matching*, Proc. 9th European Conf. Object-Oriented Prog., Aarhus, Denmark, 1995

[Bruc93] K.Bruce - *Safe type checking in a statically-typed object-oriented programming language*, Proc. ACM Symposium on Principles of Programming Languages, 1992, pag. 316-327

[BSvG95] K.Bruce, A. Schuett, R. van Gent - *PolyTOIL: A type-safe polymorphic object-oriented language*, ECOOP '95 Proceedings, LNCS 952, Springer-Verlag, pag. 27-51.

[BvG93] K. Bruce, R. van Gent - *TOIL: A new types-safe object-oriented imperative language*, Technical Report, Williams College, 1993

[CCHOM89] P. Canning, W. Cook, W. Hill, W. Olthoff, J. Mitchell - *F- bounded quantification for object oriented programming*, in Proc. Functional Programming Languages and Computer Architectures, 1989, pag. 273-280

[CHC90] W. Cook, W. Hill, P. Canning - *Inheritance is not subtyping*, Proc. 17th ACM Symp. Principles of Prog. Lang.,1990, pag. 125-135.

[CW85] L. Cardelli, P. Wegner - *On understanding types, data abstraction and polymorphism*, ACM Computing Surveys, 17(4),1985, pag. 471-521.

[Ghel93] G. Ghelli - *Recursive types are not conservative over* $F_{\leq}$, in Typed Lambda-Calculus and Applications, ed. M. Dezani-Ciancaglini, G. Plotkin, Springer Verlag, 1993

[Moto00] S. Motogna - *Formal approach to object oriented languages*, Ph.D. Thesis,"Babes-Bolyai" University, Cluj-Napoca, Romania, September 2000

Faculty of Mathematics and Informatics, "Babeş–Bolyai" University, 3400 Cluj–Napoca

E-mail address: motogna@cs.ubbcluj.ro