Modelling Inheritance in an Algebraic Specification Language[*]

Dieter Nazareth
Institut für Informatik, Technische Universität München
Postfach 20 24 20, D-8000 München 2, Germany
e-mail: nazareth@informatik.tu-muenchen.de

Abstract

This paper points at the retract problem that arises, when using subsorting[1] in a not necessarily executable algebraic specification language. We show how this problem can be circumvented in many important cases by the use of conditionally polymorphic functions. Further, we will see that this sorting facility is well-suited to model inheritance in a smart way. We give a simple example which demonstrates this approach in the specification language SPECTRUM.

1 Subsorting in a Specification Language

Statically sorted languages help to increase the perceivability of programs. Many errors can be detected and corrected at compile time. Sometimes, however, the sort system is too rigid and therefore restricts the expressiveness of the language.

One concept to weaken the strength of the sort system is to allow subsort relations between sorts. The partial ordering on the set of sorts is interpreted as set inclusion on the domains. The most well-known programming language with a subsorting concept is OBJ [1]. Subsorting allows us to apply functions to elements of a subsort of the function's parameter sort. This is an important concept to model inheritance in object oriented frameworks. Let us assume that the sort `Nat` is a subsort of the sort `Int`, written `Nat` $\subseteq$ `Int`. If we have a function `+` with the functionality `Int` $\times$ `Int` $\longrightarrow$ `Int`, then the application of `+` to a pair of natural numbers yields a well-sorted term, because the subsort relation states that each value of sort `Nat` is also a value of sort `Int`.

Another reason for the use of subsort relations is the possibility to avoid partial functions. Functions can be made total by restricting the parameter sort to an appropriate subsort. The predecessor function `pred` on natural numbers is an example for a partial function, because `pred` is undefined on `0`. The following signature shows, how `pred` can be made total by using a subsort:

```
sort PosNat, Nat; PosNat ⊆ Nat;
succ:Nat ⟶ PosNat;
pred:PosNat ⟶ Nat;
```

[*]This work is sponsored by the German Ministry of Research and Technology (BMFT) as part of the compound project "KORSO - Korrekte Software".

[1]We generally use the word sort instead of type, to avoid confusion with the notion of abstract data types.

The above signature allows us to apply the function `succ` repeatedly, e.g. `succ(succ (0))`. This expression is well-sorted, because the subsort relation `PosNat` $\subseteq$ `Nat` enables us to coerce the term `succ(0)` to the sort `Nat`. If we look at the expression `pred(pred(succ( succ(0))))` we would expect that the evaluation yields the proper value `0`. This term, however, is not well-sorted, because the sort `Nat` of `pred(succ(succ(0)))` must be coerced to `PosNat`, which is not possible in general.

In a functional programming language there is a simple solution to this problem. If a value of sort α must be coerced to a subsort β the compiler inserts a so-called retract function `r` with functionality $\alpha \rightarrow \beta$. This retract function yields a sort error at runtime, if the coercion is not possible. The above example would be changed by the compiler to `pred(r(pred(succ(succ(0)))))`. So parts of the sort check are deferred to the run-time of the program. The price one has to pay is the loss of the static sort system. But retract functions keep the language strongly sorted, because at run-time the sort errors are detected.

In a non-executable specification language, however, there is no such simple solution to this problem. Because, if the specification cannot be executed, parts of the sort check cannot be deferred to the run-time. Thus, in a non-executable language a strong sort system must be a static sort system. Of course, the problem is solved if we abandon the strong sort system, but especially for a specification language a strong sort system is important, because specification errors concerning the sort system can be detected at an early specification phase, before a time consuming development is started.

A trivial solution to the problem seems to be the insertion of a retract function `r`$:\alpha \longrightarrow \beta$ by the specifier. The function must be specified to behave like the identity function on the subsort β and is left open on all other elements of α. Usually these elements are mapped to the undefined value $\perp$, as in the following example:

```
r:Nat ⟶ PosNat;
∀x:Nat.if x=0 then r(x)=⊥ else r(x)=x endif;
```

But this is not a real solution to the problem. Firstly, one needs a lot of retract functions with respective axioms and one has to use this retract functions explicitly in the axioms. So the sort system again hinders the user in writing elegant specifications. Secondly, the problem was solved by using a partial function. But in the above example subsorting was introduced to avoid partial functions. Therefore, in a specification language the subsorting concept is not suitable to avoid partial functions like in a functional programming language.

Nevertheless, subsorting is also useful in a specification language because, like already mentioned, it is a central mechanism in object oriented approaches. All functions applicable to elements of a supersort can also be applied to elements of a subsort. Thus subsorting is a concept that supports the reuse of software which is one of the main goals in software engineering. For an important class of subsort applications we will show a smart solution to the retract problem.

2 Conditionally Polymorphic Functions

Another concept to weaken the strength of traditional sort systems is parametric polymorphism[2], which can be found in many modern languages. The most well-known polymorphic language is the functional programming language ML [2]. Therefore this kind of polymorphism is also sometimes called ML-polymorphism.

Parametric polymorphism allows a function to work uniformly on an infinite range of sorts having a common structure. This common structure is achieved

[2]Do not confuse this kind of polymorphism with the so-called ad-hoc polymorphism, which is also called overloading.

with the help of sort constructors and sort variables in the sort expression of an identifier. A typical example for a polymorphic function is the function `len` which computes the length of lists. This function is independent of the sort of the list elements. Therefore we can write one function `len` with the functionality `List` α $\longrightarrow$ `Nat` where α is a sort variable.

If we combine the subsorting facility with the parametric polymorphism we get a richer sort system which is decidable by static program analysis if a few restrictions are considered [3]. However, the principal sort of an expression, representing all valid sorts of the expression, cannot be expressed by a sort expression alone. We need a pair consisting of a sort expression and a set of subsort relations between sort expressions, called coercions. The λ-expression $\boldsymbol{\lambda}$`f.`$\boldsymbol{\lambda}$`x.f(f x)`, for example, has the principal sort `<`$(\alpha \longrightarrow \beta) \longrightarrow \alpha \longrightarrow \beta$`,` $\{\beta \subseteq \alpha\}$`>`. This means that the λ-expression belongs to any sort, which is an instance of the first component of the principal sort, with the additional condition that the respective instances of α and β must fulfill the given coercion, namely that β must be a subsort of α. In the specification language SPECTRUM [4] such functions can be declared in the signature of a specification and are called *conditionally polymorphic functions.*

In many cases this kind of functions is suitable to overcome the retract problem. Let us look at the following signature:

```
sort Nat, Int, Real;
Nat ⊆ Int, Int ⊆ Real;
.+.:Real × Real ⟶ Real;
```

It allows us to apply the function `.+.` to elements of sort `Real, Int` and `Nat`. But the result is always of sort `Real` and must be coerced with a retract function to the appropriate subsort. Now we change the functionality of `.+.` in the following way:

`.+.:` $\alpha \times \alpha \longrightarrow \alpha$ $\{\alpha \subseteq$ `Real`$\}$;

This signature also allows us to apply `.+.` to elements of the sort `Real, Int` and `Nat`. But the result of the application is now of sort `Real, Int` or `Nat`, depending on the sort of the arguments. The expression `3+4`, for example, is of sort `Nat`, if `3` and `4` are of sort `Nat`.

The conditional polymorphism is a more powerful sort concept, which enables us to express the fact, that the result of applying a function to an element of a subsort again is an element of this subsort. This is some kind of closure property. The subsorting concept by itself is too weak to express such a property.

Note, however, that this proceeding is not suitable in general to solve the retract problem in statically sorted languages. E.g. it is not possible to change the functionality of `.-.:Real × Real ⟶ Real` in the same way, because the subtraction of two natural numbers not always yields a natural number, i.e. the subtraction is not closed with respect to the natural numbers. But the closure property is typical for object oriented problems and therefore the conditional polymorphism is well-suited to model inheritance in object oriented frameworks.

3 The Specification Language SPECTRUM

We will now show on an example how the conditional polymorphism can be used to specify a system with subsort relations in a smart way. To understand the example specification we give a brief introduction to the specification language SPECTRUM we will use. A detailed informal introduction can be found in [4].

SPECTRUM is an algebraic specification language which is currently developed at the TUM[3]. The language is not restricted to conditional-equational axioms, but allows full first order logic with higher order functions. It contains explicit support for partial and non-strict functions and has a rich sort system including polymorphism and subsorting. The semantics of a specification is denoted by the class of all algebras which fulfill the given axioms[4].

We will now explain some basic syntactic constructs that are used in the example. The language is divided into two parts, one for specifying a component "in the small" and one for structuring specifications "in the large". As usual, a specification in the small consists of a signature part and an axioms part. A one line comment starts with −−. The arrow ↠ in a sort expression means that the identifier denotes a strict and total function. Each specification implicitly contains a predefined signature including the sort `Bool` and some basic functions like the polymorphic equality `.=.:`$\alpha \times \alpha \longrightarrow$ `Bool`. SPECTRUM has a similar construct like the data-construct in Haskell [5]. In the specification `CART_COORD` (see Section 4) the sort `Coord` is specified together with its constructor .⊢. and its destructors → and ↑, selecting the x-part respectively y-part of a coordinate.

In our example we only need the **enriches** operator to structure the specification. This operator is used to build hierarchical specifications by adding new sort, function symbols and axioms to a given specification. These basic explanations should be enough to understand the following example.

4 The Specification of a Graphic System

The following example deals with the specification of a graphic system, one of the first application areas of object oriented techniques. A similar specification can be found in [6], but in contrast, we will use conditionally polymorphic functions to model the operations that can be applied to the objects. We restrict ourselves to an abstract requirement specification. This specification can be used as a base for developing executable implementations.

The specification of the graphic system is based on the specification `CART_COORD`, which provides the sort `Coord` together with the constructor, the destructors and a few basic operations on coordinates. We assume that there is a specification `NUMBERS` providing the sorts `Nat`, `Int` and `Real` together with the usual operations.

```
CART_COORD = { enriches NUMBERS;

sort Coord = .⊢.(→:Int,↑:Int) strict;

.+.: Coord × Coord ↠ Coord;
.-.: Coord × Coord ↠ Coord;
mult: Coord × Nat ↠ Coord;
scale_coord: Coord × Nat × Coord ↠ Coord;

axioms ∀x1,x2,y1,y2:Int;n:Nat;c1,c2:Coord in

  (x1⊢y1) + (x2⊢y2) = (x1+x2)⊢(y1+y2);
  (x1⊢y1) - (x2⊢y2) = (x1-x2)⊢(y1-y2);

  mult(x1⊢y1,n) = (x1*n)⊢(y1*n);
  scale_coord(c1,n,c2) = c2 + mult(c1-c2,n);
  −− coordinate c1 is scaled with factor n
```

[3] Technische Universität München
[4] This is known as loose semantics.

  —— relative to the reference point c2

endaxioms; }

A graphic system generally provides a set of graphic objects together with some operations to manipulate them. In our specification `GRAPHIC_SYSTEM` we consider only the moving and scaling of objects. These two operations are specified by conditionally polymorphic functions, because they are both closed with respect to all graphic objects. For example moving or scaling a line yields again a line. If we would use a function `move'`: `Graphic_Obj` $\times$ `Coord` $\twoheadrightarrow$ `Graphic_Obj` we would loose sort information by moving objects, because the result of applying the function `move'` to a line would always be of sort `Graphic_Obj`. The same holds for the scaling function.

In our specification each object is surrounded by an invisible rectangular frame. The functions `bl` and `tr` yield the respective bottom-left and top-right coordinate of this frame. The manipulating operations are specified by describing their effect on the surrounding frame.

```
GRAPHIC_SYSTEM = { enriches CART_COORD;

sort Graphic_Obj;

bl, tr: Graphic_Obj ↠ Coord;
move: α × Coord ↠ α {α ⊆ Graphic_Obj};
scale: α × Nat × Coord ↠ α {α ⊆ Graphic_Obj};

axioms ∀ g:Graphic_Obj; c,d:Coord; n:Nat in

  (move(g,c) = move(g,d)) ⇔ (c = d);
  move(g,bl g) = g;
  move(move(g,c),d) = move(g,d);

  bl(move(g,c)) = c;
  tr(move(g,c)) = bl(move(g,c)) +
                  (tr g - bl g);

  bl(scale(g,n,c)) = scale_coord(bl g,n,c);
  tr(scale(g,n,c)) = scale_coord(tr g,n,c);

  scale(g,n,c) = move(scale(g,n,bl g),
                      scale_coord(bl g,n,c));

endaxioms; }
```

So far we have specified only those general properties of our manipulating operations that are valid for all graphic objects. In `POINT_LINE` we now add two kinds of objects, points and lines, and give additional properties for them. The sort `Point` is specified to be a subsort of `Line`, because every point is a trivial line. `Line` is a subsort of `Graphic_Obj`. Therefore all general properties of graphical objects are also valid for points and lines.

In the axioms marked with an * we profit from the conditional polymorphism. Because the universal quantification of `l` is restricted to the sort `Line`, both `move` and `scale` yield a value of sort `Line`. Therefore the function `length` can be applied without retract function. If we would use the function `move'` in the same context we would need an appropriate retract function to coerce the object from sort `Graphic_Obj` to sort `Line`.

```
POINT_LINE = { enriches GRAPHIC_SYSTEM;

sort Point, Line;
Point ⊆ Line; Line ⊆ Graphic_Obj;

length: Line ⇀ Real;

axioms ∀ p,q:Point; l,l1,l2,l3:Line;
       c:Coord; n:Nat in

  bl p = tr p;
  (∃p:Point. p=l) ⇔ length l = 0;

  (bl p = bl q) ⇔ (p = q);

  ((bl l1 = bl l2) ∧ (tr l1 = tr l2))
  ⇒ (length l1 = length l2);

  ((bl l1 = bl l2) ∧ (bl l1 = bl l3) ∧
   (tr l1 = tr l2) ∧ (tr l1 = tr l3))
  ⇒ (l1=l2) ∨ (l1=l3) ∨ (l3=l2);

  length(move(l,c)) = length l;       --*
  length(scale(l,n,c)) = n*length l;  --*

endaxioms; }
```

5 Conclusion

We pointed at the retract problem which arises when using subsort relations in a statically sorted language. We have seen that subsorting is not suited to avoid partial functions in non-executable specification languages like in the executable specification language OBJ. However, for an important class of subsorting applications the retract problem can be circumvented with conditionally polymorphic functions. We demonstrated that the conditional polymorphism is well-suited to specify with inheritance in object oriented applications. Together with parametric polymorphism the subsorting concept is therefore also useful in a non-executable algebraic specification language, because it helps to solve the problem of software reuse in software engineering.

References

[1] J.-P. Jouannaud, C. Kirchner, H. Kirchner, and A. Megrelis. OBJ: Programming with equalities, subsorts, overloading and parameterization. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *Algebraic and Logic Programming*, pages 41–53. Akademie-Verlag Berlin, 1988.

[2] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh, November 1986.

[3] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In *ESOP 88*, pages 94–114. Springer Verlag, 1988.

[4] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language

SPECTRUM, An Informal Introduction, Version 0.3. Technical Report TUM-I9140, Technische Universität München, 1992.

[5] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2).* ACM SIGPLAN Notices, May 1992.

[6] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *LNCS*. Springer, 1991.