A Universally Polymorphic Specification Language
A Brief Informal Introduction[*]

Dieter Nazareth
Institut für Informatik
Technische Universität München
Arcisstr. 21, D-80290 München
e-mail: nazareth@informatik.tu-muenchen.de

November 8, 1993

Abstract

This paper first discusses the advantages of sorts in algebraic specification languages. It gives a brief introduction into polymorphism approaches used in functional programming languages to weaken the strength of traditional sort systems. We then sketch a universally polymorphic specification language which combines all polymorphism concepts within one universal approach. The approach is based on qualified sorts which generalize the Haskell sort classes to n-ary sort predicates. To describe the properties of arbitrary sort predicates we use a separate Horn-Clause specification. In an introductory example we show how to model sort classes with our approach. A further example demonstrates that this universal polymorphism approach is also well-suited to model inheritance in algebraic specification languages.

1 Sorts in Specification Languages

Most formal languages use sorts to distinguish between different kinds of values. There are many reasons for the use of sorts. First of all sorts are an important, universal *ordering principle* and are not invented by computer scientists. It is a basic property of mankind to classify objects of the world by common features. By classifying objects one tries to put order into chaos.

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

Therefore the usage of classifications, which we call sorts, is also important in a specification language, because it forces the user to structure specifications. This decreases both the error rate and makes specifications more readable for people. Sorts can even be seen as a simple form of documentation.

Further, sorts are an important *abstraction principle*. They allow us to make propositions about a collection of objects, which are valid for each member of this collection. Thus, we can abstract in propositions from individual objects. At the same time the proposition is restricted to the objects of a sort. In axiomatic specification languages sorts serve therefore as a basis for restricting properties to a subset of the universe of objects.

Besides this classical ordering and abstraction principle there are further reasons for using sorts in specification languages, e.g. avoiding partial functions or improving performance. But at this point we do not want to go into further details.

Up to now we enumerated only advantages of sorts, but of course there are also disadvantages. Sometimes the sort system hinders the user in his flexibility and expressibility, because the sort system imposes too many restrictions on the user. In some cases a problem must be adapted to fit a specific sort system. This often leads to artificial specifications.

Another problem is that the user sometimes must write too much sort information, which is unnecessary because it can be deduced from the context. This is not only tiresome to write, but can lead to "unreadable" specifications. The aim of a language designer must therefore be to provide a sort system that restricts the user as little as possible, but as much as necessary.

However, sorts are only advantageous if their correct usage is automatically checked. A sort system only makes sense, if we can guarantee that a specification contains no error with respect to the sort system. Such sort systems are called *strong sort systems*. Only a strong sort system really helps to decrease the error rate, because sort errors are always detected and reported to the user. Because axiomatic specifications are in general not executable, the sort correctness must be checked by a static specification analysis. In the framework of algebraic specifications a strong sort system must therefore be a *static sort system*, i.e. a sort system where the well-sortedness can be determined by static specification analysis. This condition strongly restricts the choice of suitable sort systems for specification languages.

2 An Introduction to Polymorphism

Many efforts have been made to weaken the strength of traditional sort systems. Most of them try to allow a term to be not only of one sort, but of many different sorts. This ability in general is called *polymorphism*. This section gives a brief introduction into polymorphism concepts used in functional programming languages. A detailed introduction can be found in [CW85].

One of the oldest polymorphism concepts is the so-called *ad-hoc polymorphism*, better known as *overloading*. Overloading allows to use the same identifier for completely different functions. A typical example is the infix identifier + which is not only used for addition of various numbers, but sometimes also for the concatenation of lists. In a language that provides overloading the user can define both functions, i.e.

$$+ : \texttt{Nat} \times \texttt{Nat} \rightarrow \texttt{Nat}$$
$$+ : \texttt{List} \times \texttt{List} \rightarrow \texttt{List}$$

and use them both in one term. From the sort context, normally by static program analysis, it is decided which + must be used, i.e. the overloading is resolved. Strictly speaking, this is therefore not a kind of polymorphism because each term again has exactly one sort. If not, the overloading can not be resolved. Normally these terms are rejected as not well-sorted. Because overloaded identifiers are resolved, ad-hoc polymorphism does not have to be dealt with in the semantics.

The most well-known proper polymorphism concept is called, due to Strachey [Str67], *parametric polymorphism*. As the name suggests this is some kind of abstraction principle that allows us to abstract from a concrete sort in a term.

Let us consider a function `length` that takes a list as argument and returns the length of the list as result. This function can be defined by the following two equations:

```
length emptylist = 0
length(append(x,l)) = succ(length l)
```

The definition of the length function is independent of the contents of the list (represented by `x`) and therefore independent of the sort of the list elements. Thus, we can abstract from the element sort and assign the following sort scheme to the function `length`:

$$\texttt{length} : \Pi\alpha.\ \texttt{List}\ \alpha \rightarrow \texttt{Nat}$$

α is used as a sort variable and Π as universal quantification for the sort variables, indicating that α can be replaced by an arbitrary sort. `List` is a unary function, called sort constructor, on the sort level taking a sort as argument and yielding a sort as result. The parametric polymorphism allows now to apply the function `length` to a list of arbitrary sort. Let for example the variables `ln: List Nat`, `lln: List List Nat` and `lb: List Bool` be defined. Then `length` can be applied to all those variables, i.e. `length(ln)`, `length(lln)` and `length(lb)` are well-sorted terms.

If the sort quantifier is only used at the outermost level of a sort term, as in our example, we get the simplest form of parametric polymorphism, called *shallow polymorphism* or *Hindley/Milner polymorphism*, due to Hindley [Hin69] and Milner [Mil78]. Because all sort variables are bound at the outermost level, the quantifier Π is normally omitted. This kind of polymorphism can be found in many modern, functional programming languages. The first one was ML [Har86], therefore the Hindley/Milner polymorphism is sometimes also called *ML polymorphism*.

Another polymorphism concept that weakens the strength of traditional sort systems is called *subsort polymorphism* or short *subsorting*. The subsort polymorphism allows to define a partial order on sorts, called *subsort relation*, which is interpreted as set inclusion on the domains. The most well-known programming language with a subsorting concept is OBJ [JKKM88]. Subsorting allows us to apply functions to elements of a subsort of the function's parameter sort. Let us assume that the sort `Nat` is a subsort of the sort `Int`, written $\texttt{Nat}\subseteq\texttt{Int}$. If we have a function $+$ with the functionality $\texttt{Int}\times\texttt{Int}\rightarrow\texttt{Int}$, 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`.

Subsort polymorphism enables us to model inheritance in object oriented frameworks. We will demonstrate in Section 5 how to model inheritance with our universal polymorphism approach.

During the development of the functional programming language Haskell [HJW92] the developers realized, that the existing polymorphism concepts are not well-suited to model some kind of functions. They noticed that between ad-hoc polymorphism and parametric polymorphism there is in some sense a gap.

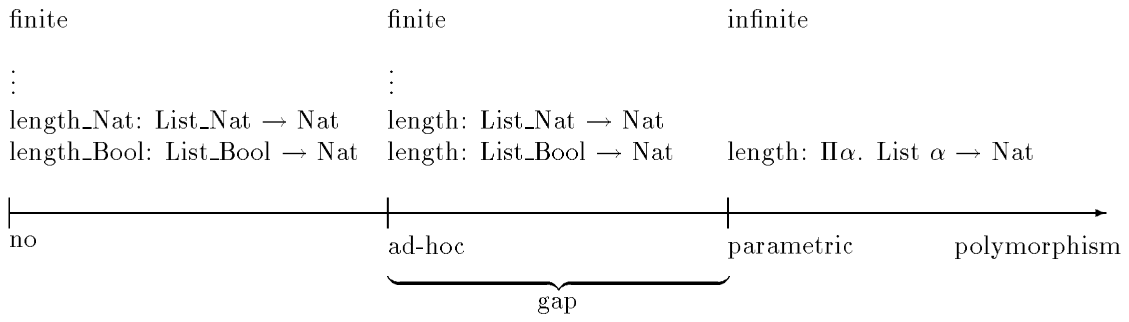

Figure 1: Polymorphism Gap

Figure 1 shows, how the length function can be handled with different polymorphism concepts. The difference between no polymorphism and ad-hoc polymorphism is only a syntactic one, because overloading only allows to use the same function identifier several times in one scope. But we are not allowed to overload sort identifiers. We have to use a different identifier for each kind of lists. Thus, in both cases we have a separate function for each sort and we can only define a finite number of length functions. The parametric polymorphism in contrast allows us to define only one length function which can be applied to lists of every sort, because the sort variable α can be instantiated by an arbitrary sort. This yields an infinite number of instantiations.

The problem is that there are a lot of functions that should be applicable to an infinite number of sorts, but not to all sorts, i.e. an infinite subset of the universe of sorts. A typical example is the equality between values. The equality function `==` should be applicable on nearly every sort, but for example not on functional sorts, because the equality of functions is undecidable. In a program the application of `==` to functions leads to a run-time error which should be avoided by the sort system. However, this can be achieved neither by ad-hoc, nor by parametric polymorphism.

In the functional programming language Haskell this problem was solved by using *sort classes*[1] [WB89]. This approach generalizes the equality type variables of ML (see [Pau92] for details) and closes the gap between ad-hoc and parametric polymorphism. It allows to structure sorts by typing them. In Haskell these types are called *classes*. The classes are used to restrict the bound sort variables in polymorphic functions by tagging them with classes. This leads to a typed abstraction mechanism. The equality function `==` can be defined in the following way[2]:

`==:` $\Pi\alpha$`::EQ.` $\alpha \times \alpha \rightarrow$ `Bool`

This means, that `==` is only available on sorts which are in the class `EQ`. With the following two definitions we can define which sorts are belonging to class `EQ`:

```
Nat::EQ
List::(EQ)EQ
```

The first line states that `Nat` must be in class `EQ`. The second line means, that `List` is a function that takes a sort of class `EQ` and yields a sort of class `EQ`. In other words, if sort `A` is in class `EQ` the sort `List A` is also in class `EQ`. These declarations allow us to apply the function `==` on an infinite, but restricted number of sorts. In our case `==` can be applied on all nested lists of sort `Nat`, i.e. `List Nat`, `List List Nat`, ..., but on no other sort.

There are a lot of other examples where we want to use a function on an infinite, but restricted number of sorts. See [HJW92] or [BFG$^+$93] for more examples.

Another reason for the introduction of the sort class concept in Haskell was to make "ad-hoc polymorphism less ad-hoc", as Wadler and Blott stated in [WB89]. In other words, the sort class concept can be used to model ad-hoc polymorphism in a more structured way. Therefore in the following we do not deal with traditional overloading anymore.

[1]In the functional community they are called *type* classes, of course.

[2]We will not use Haskell syntax in this section.

3 A Universal Approach to Polymorphism

In the last section we gave an overview of the most important polymorphism concepts. We saw that parametric polymorphism, subsorting and the sort class concept are very useful concepts to weaken the strength of a traditional sort system. They all help to describe the task of a system in a problem oriented way. It is therefore a natural desire to unify all these concepts in one language by a universal polymorphism approach.

Mark P. Jones extended the sort class concept to n-ary sort predicates [Jon92]. In his approach bound sort variables can not only be restricted by sort classes, but by arbitrary n-ary sort predicates. This allows to define functions of the following sort scheme:

$$\texttt{f: } \Pi\alpha_1,\ldots,\alpha_n.\ \texttt{P}_1[\alpha_1,\ldots,\alpha_n],\ldots,\texttt{P}_m[\alpha_1,\ldots,\alpha_n] \Rightarrow \tau[\alpha_1,\ldots,\alpha_n]$$

τ is a sort expression containing possibly the sort variables $\alpha_1,\ldots,\alpha_n$. The sort variables can be restricted by predicates $\texttt{P}_1,\ldots,\texttt{P}_m$ of arbitrary arities. The , between the predicates is interpreted as logical $\wedge$. Jones calls these restricted sorts *qualified sorts*. These qualified sorts are the basis for a universal approach to polymorphism, because the above sort scheme includes all sort expressions needed for the polymorphism concepts mentioned in the last section as a special case. The functions from the last section can be given the following, in some sense equivalent qualified sorts:

$$\texttt{length: } \Pi\alpha.\ \texttt{List}\ \alpha \rightarrow \texttt{Nat}$$
$$\texttt{==: } \Pi\alpha.\ \texttt{EQ}(\alpha) \Rightarrow \alpha \times \alpha \rightarrow \texttt{Bool}$$

Parametric polymorphism can be expressed trivially by qualified sorts, because we do not need any restricting predicate. Therefore the length function stays the same. Also for the sort class concept it is quite easy to see that it fits into the above scheme, because a class can also be seen as a unary sort predicate. In our example the class `EQ` is replaced by a sort predicate `EQ`.

More complicated is the handling of subsort polymorphism. In Section 2 we presented an *implicit* subsorting concept. Implicit, because besides the subsort relation we do not need any other concept to use the subsorting, i.e. we do not need any kind of *coerce* function, coercing elements of a sort to elements of a supersort of this sort. The coercion is done implicitly, because the subsort relation is interpreted as set inclusion on the domains. This implicit subsorting, however, can not be expressed with qualified sorts. We can only provide some kind of *explicit* subsorting.

Let us look at the addition function + from Section 2. In a first step we abstract from the concrete sort `Int` and get the following function:

$$+\texttt{: } \Pi\alpha.\ \alpha \times \alpha \rightarrow \alpha$$

However, this function is too general, because now + can be applied to elements of any sort. Thus, we must restrict the sort variable α by an appropriate predicate. We only want to apply + on subsorts of `Int`. This can be achieved by the following function:

$$+ : \Pi\alpha.\ \alpha \subseteq \texttt{Int} \Rightarrow \alpha \times \alpha \rightarrow \alpha$$

The binary sort predicate $\subseteq$ is used to restrict the sort variable α to all subsorts of `Int`. This is, however, only the intended semantics behind the purely syntactic identifier $\subseteq$. Later we will see, how to define this semantics.

Qualified sorts offer only explicit subsorting, because every function that wants to use the subsort relation must explicitly use the restricting predicate in its functionality. But this is not a drawback , because implicit rules are always more susceptible to errors. Explicit subsorting achieved by qualified sorts is even more expressive, because we can express the dependence of the result sort on an argument sort. In our example adding two natural numbers yields again a natural number and not an integer as in the implicit version. With implicit subsorting this can only be achieved by overloading the identifier + or by using so-called *retract* functions. See [Naz93] for a closer look at the problems that arise, when using subsorting in an algebraic specification language.

Besides sort classes and subsorting there are other interesting applications of qualified sorts. See [Jon92] for more examples.

As part of his thesis Jones occupied himself with the syntactic treatment of qualified sorts in functional programming languages. He extends the Damas/Milner approach [DM82] to type inference and gives an algorithm that infers the most general sort in the presence of qualified sorts. In his work he abstracts from a concrete predicate system. His results are, up to a few basic assumptions, independent of a concrete predicate system.

But qualified sorts are only a basis for a universally polymorphic language, because we need a framework to describe the properties of the predicates used in the qualified sorts. As already mentioned, $\subseteq$ is only a predicate identifier without any semantics. We must fix some properties like reflexivity and transitivity, and we must define the desired subsort relations, like $\texttt{Nat}\subseteq\texttt{Int}$. The same holds for the class predicates. To complete the sort class concept, we must be able to define, which sorts belong to a class. For example, we must be able to define, as in Section 2, that all nested lists of sort Nat belong to class `EQ`.

But we do not want to restrict our specification language to the polymorphism concepts described in Section 2. Therefore we need a universal predicate language that allows us to define the properties of arbitrary predicates used in qualified sorts.

4 A Universally Polymorphic Specification Language

In this section we want to sketch briefly the idea of a universally polymorphic specification language, i.e. a specification language that provides qualified sorts together with a universal language to specify the properties of sort predicates. We do not want to go into syntactic details and we do not want to speak about the semantics of such a language. These areas are treated in [Naz94].

To realize this idea, we need a two-leveled specification language. Our specifications are divided into a specification on the sort level, called *sort specification*, and a specification on the object level, called *object specification*, where the second one depends on the first one. The specification on the object level as usual defines the behaviour of the constants and functions. The sort level specification defines the properties of the sort predicates used in the qualified sorts of the object level.

On the object level we use a classical first order logic with higher order functions, e.g. LCF [Pau87]. Because of the sort specification being used to define the well-sortedness of the object specification, the sort specification must in some sense be executable. Therefore we use a Horn-Clause logic on the sort level. This language is on the one hand general enough to describe the properties of arbitrary predicates and is on the other hand executable via resolution.

As already mentioned, we do not give an exact definition of our language. We will explain the basic setup on a simple example which uses sort classes. In Section 5 we will show on a larger example how to model inheritance with this approach.

```
Example = {

    -- Specification on the sort level
    -- Sort constructors
    cons Bool₀; Nat₀; List₁;

    -- Sort predicates
    pred EQ₁;

    -- Horn-Clause axioms
    sortaxioms
        EQ(Nat);
        EQ(α) ⇒ EQ(List α);
    endaxioms

    -- Specification on the object level
    -- Functions
    fun  0: Nat;
```

```
        succ: Nat → Nat;
        emptylist: Πα. List α;
        append: Πα. α × List α → List α;
        #: Πα. EQ(α) ⇒ α × List α → Nat;
        .==.: Πα. EQ(α) ⇒ α × α → Bool;

    -- First-order axioms
    axioms EQ(α) ⇒ ∀ a,b,c: α, l: List α.

        -- Laws for ==
        (a==a) = true;
        (a==b) = true ∧ (b==c) = true ⇒ (a==c) = true;
        (a==b) = true ⇔ (b==a) = true;

        -- Laws for #
        #(a,emptylist) = 0;
        #(b,append(a,l)) = if a==b then succ(#(b,l))
                                   else #(b,l)
                           endif;

        -- Application of # (theorem)
        #(0,append(0,emptylist)) = succ 0;

        -- Further laws
        ⋮

    endaxioms }
```

-- starts a one line comment. The specification is split into a sort level and an object level. As usual in the framework of algebraic specifications both levels consist of a signature and an axioms part.

On the sort level the signature consists of the sort functions, called *sort constructors*, and the sort predicates. The number assigned to each identifier expresses the arity of the constructor respectively predicate. In our example we define a zero-ary sort constructor `Nat`, i.e. a sort constant or *basic sort* and a unary sort constructor `List`. Further we define a unary sort predicate `EQ`.

In the axioms part of the sort level we define the desired properties of our predicate `EQ`. Speaking in terms of classes, the first axiom states that sort `Nat` belongs to class `EQ`. The second axiom says that if a sort α belongs to class `EQ` then sort `List` α also belongs to this class. Thus we have defined the same behaviour as in Section 2. This is of course only a small example. Our Horn-Clause logic is universal enough to express more complicated structures, as we will see in Section 5.

On the object level we will now use the predicate `EQ` to restrict our polymorphic functions. In the signature part of the object level we declare our functions. This is the only place where we can declare a polymorphic function. If we allowed to define a polymorphic identifier by λ-abstraction, we would get a higher order function with respect to polymorphism, i.e. we would get a function that takes a polymorphic function as argument. This is not allowed in our approach. The reason is that we restrict our language to shallow polymorphism. Therefore it also makes no sense to allow polymorphic λ-abstractions, because this λ-abstraction can only be applied once and not passed as argument.

We also do not allow to define polymorphic functions by the quantifiers $\forall$ and $\exists$. The reason is that we do not want to make propositions about a set of polymorphic objects. We only want to describe the behaviour of a particular polymorphic object, like `==`. Another more technical reason is that the sort of bound identifiers is automatically inferred by a sort inference system, i.e. the user can omit sort annotations for local identifiers. But if these identifiers are allowed to be polymorphic no sort error would ever be detected, because the system can always infer the totally polymorphic sort $\Pi\alpha\,.\,\alpha$ for local identifiers. This would lead to a nearly unsorted language. Thus, the signature of a specification replaces the let-construct used in functional languages to define polymorphic functions.

On the object level we now define a polymorphic equality predicate `==` where the sort variable α is restricted by the sort predicate `EQ`. In the same way we restrict a function `#`, counting the number of occurrences of a particular element in a list. Both functions are therefore only applicable on sorts that fulfil the sort predicate `EQ`. Looking at the sort specification these are just all nested lists over `Nat`, i.e. the sort specifications gives the intended semantics to `EQ`. Besides these two functions which are the focus of our attention we need constructors for our sorts `Nat` and `List` α with the usual semantics. We assume that all functions in the signature are strict.

In the axioms part we now describe the laws of our functions by using traditional first order logic. We assume that $\forall$ quantifies only over defined values. The laws for `==` as well as for `#` are as usual. The equality function `==` is defined to be an equivalence relation and the counter function `#` is defined recursively. The interesting point of the specification lies in the universal quantification of the variables used to define the polymorphic functions. These variables are assigned the sort α respectively List α. But note that by doing this we do not declare a polymorphic identifier. In both cases the sort variable α is not bound by a Π at the outermost level of the sort expression, which is not allowed for local identifiers. Thus, the variables have the same sort at each occurrence in the axioms and are therefore not polymorphic. All sort variables are automatically universally quantified at the outermost level of the object specification.

Like in the signature, we must restrict the sort variable α by the sort predicate `EQ`. Otherwise the specification would not be well-sorted, because `==` as well as `#` can only be applied to variables whose sort fulfil the predicate `EQ`. By using sort variables in the axioms part we can fix the laws not only for one sort. The laws must be valid for all instantiations of the sort variables which fulfil the restricting predicate.

To show the application of # we added a simple theorem to our specification. In the theorem we apply # to a list of natural numbers. This application is well-sorted, because we can deduce from the sort specification that `List Nat` fulfils the sort predicate `EQ`.

The example shows quite well the connection of polymorphism to parameterized specifications. Without polymorphism concept we would have used a parameterized specification for that problem. In [GN93] it is shown that in many cases parameterized specifications can be replaced by more elegant polymorphic specifications.

5 Modelling Inheritance

In the last example we only gave a simple application of our universally polymorphic specification language. The example can, in a similar way, already be written in the specification language SPECTRUM [BFG+93], which provides a sort class concept. The difference, however, is that SPECTRUM only provides sort classes, whereas in this approach sort classes are only one of many possible applications. In this section we will give an example of a completely different application of our universal polymorphism approach. We will show that our language is also well-suited to model inheritance in a smart way.

The example deals with the specification of a graphics system, one of the first application areas of inheritance techniques. A similar specification can be found in [Bre91], which uses an implicit subsorting concept with overloading to model the operations that can be applied to graphic objects. We restrict ourselves to an abstract requirement specification. This specification can be used as a basis to develop executable implementations.

In our example we use an operator `enriches` to structure the specification. This operator is used to build hierarchical specifications by adding new sort constructors, sort predicates, functions and axioms to the respective part of a given specification.

The specification of the graphics system is based on a specification `CART_COORD`, which provides the basic sort `Coord` together with the constructor, the selectors 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. Moreover, we assume that ∀ quantifies only over defined values and that all functions are strict.

```
CART_COORD = { enriches NUMBERS;

     cons Coord$_0$;
fun   −− Constructors and Selectors for Coord
      .⊢.: Int × Int → Coord;
      →: Coord → Int;
      ↑: Coord → Int;

      −− Other functions on Coord
      ⊕: 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.

      −− Laws for constructor and selectors
      →(x1⊢y1) = x1;
      ↑(x1⊢y1) = y1;
      →c1 ⊢ ↑c1 = c1;

      −− Laws for other functions
      (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
      −− relative to the reference point c2

endaxioms }
```

A graphics 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 should be available on all kinds of objects. Therefore we use polymorphic functions and restrict the operations to graphic objects. Graphic objects are typically hierarchically ordered by a so-called *is-a* relation. For example, a point is a line and a line is a rectangle and a rectangle is a graphic object. This is-a relation is used to specialize objects. A graphic object in our example is the most vague term with the least properties. The terms rectangle, line and point are more and more concrete and are only special cases of the others.

The is-a relation is strongly connected with the concept of inheritance. A specialized object inherits all properties from a more general object. For examples points, lines and rectangles inherit all general properties of a graphic object. These properties also include the general operations like moving and scaling.

Normally the is-a relation is modelled by a subsort relation. Our universal approach, however, allows us to specify directly this is-a relation. This leads to a very problem-oriented specification and shows that our language is flexible enough to describe different kinds of problems in a natural way.

On the sort level we define a new basic sort `Graphic_Obj` and a binary infix sort predicate `.is−a.`. In the axioms part of the sort specification we define the laws of the new sort predicate, namely the reflexivity and the transitivity.

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. All functions are defined polymorphically and restricted by the predicate α `is−a Graphic_Obj`. All functions can therefore be applied only on sorts which are graphic objects. We do not go into each single axiom. Again we only look at the universally quantified identifiers.

The properties described in this specification are very general and should be valid for all graphic objects. Therefore we use the sort variable α for the identifier `g` and restrict α by the predicate α `is−a Graphic_Obj`. Note that giving the identifier `g` the sort `Graphic_Obj` would yield a completely different semantics. In this case all axioms would only be valid for the elements of sort `Graphic_Obj` and not e.g. for the elements of sort `Line`, defined in the next specification. This is because our approach is explicit. Defining sort predicates together with rules does not influence the domains, i.e. the sort predicate `is−a` does not impose a subset order on the domains.

```
GRAPHIC_SYSTEM = { enriches CART_COORD;

     cons Graphic_Obj₀;

     pred .is−a.;

     −− Laws for the subsort predicate is−a
     sortaxioms
          α is−a α;                          −− Reflexivity
          α is−a β, β is−a γ ⇒ α is−a γ;  −− Transitivity
     endaxioms

     fun  bl, tr: Πα. α is−a Graphic_Obj ⇒ α → Coord;
          move: Πα. α is−a Graphic_Obj ⇒ α × Coord → α;
          scale: Πα. α is−a Graphic_Obj ⇒ α × Nat × Coord → α;

     axioms α is−a Graphic_Obj ⇒ ∀ g: α, c,d: Coord, n: Nat.

          (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. In the axioms part of the sort level we now define that `Point is−a Line` and that `Line is−a Graphic_Obj`. Therefore all general properties of graphic objects are inherited by points and line.

For all objects that are lines we define an additional function `length` computing the length of a line. Again we use a restricted sort variable α for the line variables is the axioms part. Note that for describing the properties of the points we do not use a sort variable. The identifiers `p` and `q` are declared to be of sort `Point`. This is equivalent to β with the restriction β `is−a Point` because there is no proper specialization of the sort `Point`. However, if we later would like to specialize the sort `Point`, we would have to use a restricted sort variable for the identifiers `p` and `q`.

```
POINT_LINE = { enriches GRAPHIC_SYSTEM;

cons Point₀; Line₀;

sortaxioms
    Point is−a Line;
    Line is−a Graphic_Obj;
endaxioms

fun  length: Πα. α is−a Line ⇒ α → Real;

axioms α is−a Line ⇒ ∀ p,q: Point, l,l1,l2,l3: α, c: Coord, n: Nat.

    bl p = tr p;
    length p = 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 }
```

6 Conclusion

We have sketched a universally polymorphic specification language, which allows to use all kinds of polymorphism in one general approach. This is in contrast to other languages which provide only one or two different kinds of polymorphism, e.g. the specification language SPECTRUM which provides only parametric polymorphism and sort classes. We have seen in different examples that our approach is flexible enough to model sort classes as well as inheritance in a very direct way. Thus, the sort system does not hinder the user, but supports him in writing problem-oriented specifications. Our approach is therefore a big step towards the fulfilment of the requirement, that a sort system should restrict the user as little as possible, but as much as necessary.

References

[BFG+93] Manfred Broy, Christian Facchi, Radu Grosu, Rudi Hettler, Heinrich Hussmann, Dieter Nazareth, Franz Regensburger, Oscar Slotosch, and Ketil Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part i. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, Fakultät für Informatik, TUM, 80290 München, Germany, May 1993.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *LNCS*. Springer, 1991.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[DM82] L. Damas and R. Milner. Principle Type-Schemes for Functional Programs. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[GN93] R. Grosu and D. Nazareth. Towards a new way of parameterization. Submitted to the Third Maghrebian Conference on Software Engineering and Artifical Intelligence, June 1993.

[Har86] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh,Department of Computer Science, November 1986.

[Hin69] R. Hindley. The Principle Type-Scheme of an Object in Combinatory Logic. *Trans. Am. Math. Soc.*, 146:29–60, December 1969.

[HJW92] 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.

[JKKM88] 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.

[Jon92] Mark P. Jones. Qualified types: Theory and practice. Technical Monograph PRG-106, Oxford University Computing Laboratory, Programming Research Group, July 1992.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Naz93] Dieter Nazareth. Modelling inheritance in an algebraic specification language. In Jianping Wu et al., editor, *Proceedings of the Third International Conference for Young Computer Scientists, Beijing*, pages 9.05–9.08. Tsinghua University Press, July 15–17 1993.

[Naz94] Dieter Nazareth. *A Universally Polymorphic Specification Language*. PhD thesis, Technische Universität München, 1994. to appear.

[Pau87] L.C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[Pau92] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[Str67] C. Strachey. Fundamental Concepts in Programming Languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, 1967.

[WB89] Philip Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.