Polymorphic Components for Monomorphic Languages

Franz-Josef Grosch and Gregor Snelting
Arbeitsgruppe Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17, D-33 Braunschweig

Abstract

Most procedural programming languages, due to their restricted type systems, do not allow for polymorphic software components in the style of functional languages. Such polymorphism however greatly increases the potential for component reuse, while still guaranteeing the security of strong typing. In this paper, we show how to obtain polymorphic software components for "ordinary" languages like C or Modula-2. Our method is based on generic type inference in a software component library. The source of polymorphism is the use of free (undeclared) names in a component. The analysis algorithm will infer signature schemes*, which are analoguous to type schemes in functional languages. Signature schemes can be used to check library consistency and allow to retrieve components by usage patterns.*

1 Introduction

Polymorphism, as introduced in the functional language ML [9], is one of the most fruitful approaches to increase software reusability. ML polymorphism allows to define functions which can be fed with arguments of different types, thereby providing for software components which can be used in different contexts, while at the same time guaranteeing the security of strong typing. The possible usage contexts of a polymorphic function are described by a *type scheme*, and theory guarantees that for correct programs, such type schemes always exist (*principal type property*), and that uses of a polymorphic function will not produce any runtime errors due to type mismatches (*well-typed programs can't go wrong*) [6]. Another advantage of polymorphic type inference is that type schemes can be used as search keys for component retrieval [10,11,12].

Unfortunately, this kind of polymorphism is not available for ordinary procedural languages. Thus, components written in e.g. Modula-2 or C must be either monomorphic — as in classical libraries —, thereby hampering the potential for being reused. Or one has to use special tricks, thereby corrupting readability and type safeness. As an example of the latter situation, consider the generic Modula-2 programs presented in [18], [5] or [3]. Here, the restrictions of the type system are circumvented by resorting to untyped pointers or byte arrays. This results in rather unreadable programs, which even might produce runtime errors since the type system has been fooled.

The programming language ADA improves on older procedural languages by offering a limited kind of polymorphism, namely *generic packages* or procedures. Generic ADA components are safe and reusable in various contexts. The language C++ offers a similar feature, namely program *templates*. But not every programmer wants to use ADA or C++.

In this paper, we show how generic type inference can be used to obtain polymorphic software components for monomorphic languages. The source of polymorphism is the use of free (i.e. undeclared) identifiers in a software component, such as a Modula-2 module or a list of declarations in C. Despite the free identifiers, type inference can check consistency of a component, and will compute what we call a *signature scheme*. A signature scheme describes exactly the possible usage contexts of a component. Signature schemes are stored in the library together with the components themselves, thus components need not be reanalyzed in order to check correct usage. Furthermore, signature schemes can be used for component retrieval based on usage patterns, and improve configuration management.

Our approach offers the everyday programmer increased reuse potential, without changing his favourite language or resorting to other unnatural devices. The method was already sketched in [15]. Meanwhile, it has been implemented as part of the inference–based software environment NORA*. This paper gives a detailed account of the underlying principles and algorithms.

2 An overview of NORA

The experimental software development environment NORA aims at utilizing inference technology for software engineering tools. Many new results and algorithms have been developed in the field of deduction sytems and unification theory, but their potential for software development

* NORA is a drama by the Norwegian writer H. IBSEN. Hence, NORA is no real acronym.

environments is far from being exhausted – on the contrary, software engineers and inference people often look at each other with displeasure.

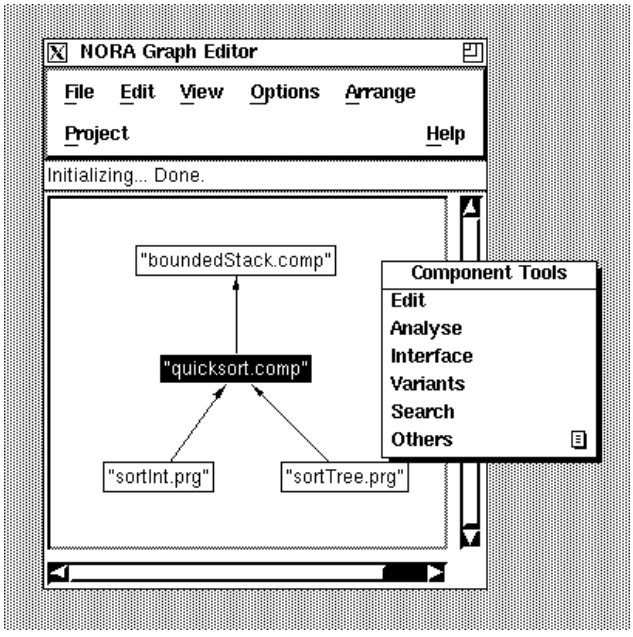
NORA deals with the following aspects of software development:

- interface checking in a library of polymorphic, reusable software components
- unification-based component retrieval, utilizing usage patterns as search keys
- interactive configuration management based on an inference engine for feature logic
- reverse engineering support by infering variant / configuration structures from existing source code.

The first aspect is the topic of this paper. In contrast to existing tools, we aim at interactive tools which are parameterized with language-specific knowledge and can handle incomplete or inconsistent information.

The picture below gives a very small example of a component libray, as displayed by NORA. The four Modula-2 components constituting this library will be used as examples later in this paper. The dependency graph is computed by language-specific rules, hence no specification (e.g. a makefile) is needed. The graph display is based on the Sugiyama algorithm [17], but the user can also manipulate the graph by himself. The whole state of a NORA session can be stored persistently.

Software components can be available in form of ASCII-files, abstract syntax trees, or even syntax trees together with semantic information. Upon selection of a component, several tools can be activated by selecting an item in the "Component Tools" menu (see figure). A component may be edited (construction and modification of components lies outside the scope of NORA, thus everybody may use his favourite editor). A component may also be syntactically or semantically analysed. The analysis algorithm, as described in this paper, will not only compute interface and signature scheme of a component, but will also check inter-component consistency. Interface information about a component can be displayed on request. If inconsistencies inside a component or between components are detected, the corresponding node resp. edge in the dependency graph is highlighted; detailed error messages can be displayed as well.



Other subsystems of NORA are the configuration management subsystem and the retrieval subsystem, which will be described elsewhere.

3 The library analysis algorithm

3.1 Software components in NORA

A traditional UNIX technique for reuse of software components is the use of the preprocessor. One reuse mechanism is based on the "#include" statement. For example, use of the regular expression matching package "regexp(3)" requires that a file is included which not only contains declarations, but also a lot of C code. Another popular technique is the use of macros via "#define", which not only provide for another reuse mechanism, but – together with "#ifdef" – also allows quite sophisticated configuration management. The drawbacks of using the C preprocessor are however well known: first, arbitrary text may be included or defined as the body of a macro, which may lead to syntactically erroneous programs after preprocessing; furthermore, include files or macro bodies cannot be typechecked in isolation and must be reanalysed upon every usage; worse, an include file or macro body may contain free variables, which are incorrectly bound at the usage site, or a macro body may contain local variables, whose names clash with actual macro parameters.

In order to overcome the drawbacks of naive preprocessing, Kohlbecker et al. developed what they call "hygienic macros" [8]. An efficient implementation of hygienic macros was later presented in [4]. Clinger's algorithm uses a clever technique of automatic variable renaming in order to avoid unwanted bindings.

Aware of these developments, and motivated by the popularity of "#include"-d software, we decided that an improved "#include" mechanism may serve very well as the basis for reusable software components written in monomorphic languages. As an example, consider the following component, which is a generic stack module written in Modula-2 (although our method is language independent, we will use Modula-2 as a focus of our investigations):

```
MODULE boundedStack;
  IMPORT elemType, stackSize;
  EXPORT initStack, push, pop, isEmpty, isFull;
  TYPE
    fixedStack = ARRAY [1 .. stackSize] OF elemType;
  VAR
    theStack: fixedStack;
    topOfStack: INTEGER;

  PROCEDURE initStack;
  BEGIN
    topOfStack := 0
  END initStack;

  PROCEDURE push(elem: elemType);
  BEGIN
    topOfStack := topOfStack + 1;
    theStack[topOfStack] := elem;
  END push;

  PROCEDURE pop(): elemType;
  BEGIN
    topOfStack := topOfStack - 1
    RETURN theStack[topOfStack + 1];
  END pop;

  PROCEDURE isEmpty(): BOOLEAN;
  BEGIN
    RETURN topOfStack = 0
  END isEmpty;

  PROCEDURE isFull(): BOOLEAN;
  BEGIN
    RETURN topOfStack = stackSize
  END isFull;
END boundedStack;
```

This local module is “out of context”, hence the two identifiers “elemType” and “stackSize” are undeclared – they are the source of polymorphism and reusability, since the code is valid for *any* “elemType” and “stackSize”.

A compiler would not be able to process such a module in isolation. NORA however allows to store this module as a reusable component, let’s say with the name “boundedStack.comp”. After syntactic analysis, type inference will compute a signature scheme for the module. In particular (as explained in detail below), analysis infers that “elemType” must be a non-structured type and that “stackSize” must be of type “INTEGER or CARDINAL”. This information is stored together with the component and can later be used to check for correct component use, to display the possible usage contexts and other information, or to search functions by type schemes.

NORA’s include mechanism will allow to reuse the module at different sites:

```
#include boundedStack.comp
```

This looks like classical preprocessing; in particular, name clashes or unwanted bindings as described above might occur. In order to avoid such binding problems, all free identifiers in the component are considered *component parameters* and may be explicitely bound at the point of inclusion (in Modula-2, not only the free identifiers are implicit component parameters, but all identifiers which are visible in the in the scope surrounding the component):

```
#include boundedStack.comp (stackSize = maxStack,
                              elemType = CARDINAL)
```

If such explicit parameter binding is omitted, the usage context is used for binding of component parameters; this is called the default parameter mechanism.

Summarizing our discourse, we define:
A NORA component is a piece of program text which

- is stored as an ASCII file, perhaps with syntax tree and semantic information attached
- constitutes a syntactic unit
- may contain free variables, which are the source of polymorphism
- has the free variables as implicit parameters
- can be reused by “#include”
- can be instantiated with actual parameters upon inclusion
- can be semantically analysed via generic type inference
- has a persistent signature scheme attached in order to check correct usage, avoid reanalysis for every use, and allow for retrieval based on type schemes as search keys.

3.2 Library dependencies

Before any semantic analysis is possible, the dependencies between components have to be computed. We are able to distinguish two sorts of library components, namely polymorphic components which are based on the language independent include-mechanism and — as Modula-2 serves as example language — definition and implementation modules which represent the Modula-2–specific software components. Although the focus of this paper lies on the language-independent polymorphic components, we also want to sketch the interplay with language-specific, monomorphic components.

As the whole analysis is more or less syntax directed, we define a simplified syntax for the essential parts needed within the algorithms. The language-independent parts are integrated into the syntax, these are “polymorphic_component” which is constrained to be a file of “declarations” or “statements” and “include” for the use of polymorphic components. The restriction of polymorphic components to “declarations” or “statements” is rather arbitrary and can be changed as needed, since the analysis is parameterized with a formal definition of syntax and inference rules.

library_component ::= *polymorpic_component* | *module*
polymorphic_component ::= *declarations* | *statements*
module ::= *Definition Module Id imports declarations* |
Implementation Module Id imports body |
Module Id imports body
imports ::= *Import Id* | *Import Id* ; *imports*
body ::= *declarations Begin statements End* ;
declarations ::= *declaration declarations* | *declaration*
declaration ::= *include* | ...
statements ::= *statement statements* | *statement*
statement ::= *include* | ...
include ::= *#Include Id* (*parameters*)
parameters ::= *formal_name* = *actual_name* |
formal_name = *actual_name* , *parameters*

We now present the dependency analysis "D" in a denotational style. Called with one library component as the first parameter, the dependencies originating from this component are computed. The parameters of the algorithm are: syntax tree, actual component "c" and a list of directed dependencies "l". It is mainly a recursive descend over the dependencies indicated within the components. Remember that language-specific and language-independent dependency rules are mixed. The function "getSyntaxTree" loads the syntax tree from the appropriate file. There are three kinds of dependencies: inclusion of polymorphic components, import of definition modules, and the dependency between an implementation module and its definition module.

D ⟦*Definition Module Id imports body*⟧ $c\ l =$
D ⟦*body*⟧ *Id* (D⟦*imports*⟧ *Id* l)
D ⟦*Implementation Module Id imports body*⟧ $c\ l =$
LET $l_1 = D$ ⟦*getSyntaxTree Id*⟧ *Id* l
IN (*c implements Id*)
^(D ⟦*body*⟧ *Id* (D ⟦*imports*⟧ *Id* l_1))
D ⟦*Module Id imports body*⟧ $c\ l =$
D ⟦*body*⟧ *Id* (D ⟦*imports*⟧ *Id* l)
D ⟦*declarations Begin statements End*⟧ $c\ l =$
D ⟦*statements*⟧ c (D ⟦*declarations*⟧ $c\ l$)
D ⟦*#Include Id* (*parameters*)⟧ $c\ l =$
(*c includes Id*)^(D ⟦*getSyntaxTree Id*⟧ *Id* l)
D ⟦*Import Id imports*⟧ $c\ l =$
LET $l_1 = D$ ⟦*getSyntaxTree Id*⟧ *Id* l
IN (*c imports Id*)^(D ⟦*imports*⟧ $c\ l_1$)
D ⟦*Import Id*⟧ $c\ l =$
LET $l_1 = D$ ⟦*getSyntaxTree Id*⟧ *Id* l
IN (*c imports Id*)^l_1
... for other nodes descend recursively; terminals simply return the accumulated dependency list

3.3 Context Relations

For the generic type inference of a single library component, we use *context relations*. Originally context relations have been developed for semantic analysis of incomplete program fragments; they have successfully been used for several languages. Context relations are described in detail in [1] and [14]. Here we only want to give an intuition how relational analysis works.

Relational analysis is similar in spirit to type inference in functional languages. It associates sets of attributes with nodes of a given abstract syntax tree. Attributes are terms of a free term algebra with variables, functors and constants. In contrast to Damas-Milner type inference where only one sort of variables is used, all variables in relational analysis are *sorted* in order to express context contraints. A syntax-directed inference system determines the computation of attributes. Again there is a difference to classical type inference: while type inference allows only one inference rule per node in the abstract syntax tree, relational analysis allows several rules per node. The main operation for infering semantic information is (sorted) *unification* of attributes.

Relational analysis is *generic*, i. e. the structure of the attributes, the sort hierarchy and the inference rules serve as language-specific parameters for the analysis algorithm. The language definer specifies the type constructors according to the language of interest. He also defines the sort hierarchy, which allows to express constraints like "the expression has a non-structured type" or "the variable must not be a control variable". Two terms are unifiable if their sorts have a common subsort. To guarantee uniqueness the sort hierarchy must be a semilattice. Typically a few syntactic constructs of procedural languages allow more than one correct typing, they are called *overloaded*. Therefore it is possible to define more than one inference rule per construct.

Before relational analysis actually starts, a *name resolution* process uniquely renames all occurrences of identifiers that denote the same object. This is necessary since we want to cope with incomplete components and undeclared identifiers. Type inference "collects" type assumptions by unifying type terms belonging to the same syntactic objects. The type assumptions are represented as a map from identifiers to type terms. In relational analysis, different attributations have to be considered due to overloading, therefore a context relation which is similar to a relation in relational data bases is used to represent possible attributations – one row represents one attributation. Inference rules are also represented as relations, so-called *basic relations*, where the columns are named by the nodes of the abstract syntax tree. Relational analysis "collects" attributes by *joining* relations. The term *join* is used due to the fact that this operation is similar to the natural join known from rela-

tional data base theory; all possible row combinations are built and the attributes in common columns are unified. If a unification fails, the corresponding attributation is wrong and the whole row is deleted. A component is semantically correct, if one or more rows survive in the result relation.

In order to describe the extension of relational analysis from single components to a whole library with polymorphic components we have to recapitulate some notions from [14].

Definition.
1. $r \sqcap s$ is called the *join* of two context relations r and s.
2. $\bigsqcap$ is the generalization of $\sqcap$ to a set of context relations.

The context relation associated to a terminal T of the abstract syntax is simply a basic relation. The context relation $cr[\![t :: t_1, ...t_n]\!]$ of a syntactically correct piece of program with root node t in the abstract syntax tree and children $t_1...t_n$ is given by the following simple formula:

$$cr[\![t :: t_1, ...t_n]\!] = basicrelation[\![t]\!] \sqcap \bigsqcap_{i=1}^{n} cr[\![t_i]\!]$$
$$cr[\![T]\!] = basicrelation[\![T]\!]$$

3.4 Library analysis

Now we are going to describe how the library analysis works. As explained above, the result of the analysis of one library component is a context relation representing its interface or *signature*. Since we are able to analyse incomplete components, even signatures of monomorphic components may contain non-ground attributes. Signatures of monomorphic components contain those objects which are not local to the analysed component. Every occurrence of such an object refines its attributation, i. e. variables in the attributation get instantiated. Objects from monomorphic components exactly have one final attributation.

The analysis of a polymorphic component results in a relation where the columns are named by those identifiers that are visible in a surrounding scope. Such a component is polymorphic, if more than one attributation is still possible, or attributes are not ground, i. e. still contain variables. This signature is turned into a *signature scheme* by universally quantifying over all variables. There are no type assumptions for polymorphic components, therefore universal quantification of all variables in the result signature is allowed (note that according to Damas-Milner, variables occuring free in the type assumptions may not be universally quantified). The analysis of "boundedStack" (see section 3.1) results in the following signature scheme. Attributes are written in a Prolog-like term notation; variables are sorted:

$\forall\ \alpha : non\_structured\_type,$
$\beta : expression,$
$\gamma : integer\_or\_cardinal.$
[elemType: $object(type, \alpha)$,
stackSize: $object(\beta, \gamma)$,
push: $object(procedure, proc([value\_parameter(\alpha)]))$,
pop: $object(procedure, func([], \alpha))$,
initStack: $object(procedure, proc([]))$,
isFull: $object(procedure, func([], bool))$,
isEmpty: $object(procedure, func([], bool))$*]*

If a polymorphic component is included in another component, the analysis has to instantiate its signature scheme. This instantiation has two aspects. First, the name resolution uniquely renames the objects in the signature scheme according to scope rules in the context. If a component is included more than once, in general different bindings will result. Second, the library analysis generically instantiates the signature scheme in order to allow different instantiations of the attributes in different contexts. This latter process is done by the function "instantiate", and can be defined as a transformation of a signature scheme in three steps:

– replace ∀-bound variables by new variables of the same sort, thereby removing the quantifiers
– rename object names that are formal parameters by the actual parameter names
– rename all objects according to the results of name resolution.

The resulting signature is used as a context relation for the analysis of an include, as shown below.

Simple analysis as described in the previous section is only performed for library components that do not depend on others. These are definition modules, program modules or polymorphic components that do not import other modules and that do not include polymorphic components. For other components, the signatures resp. signature schemes for the components they depend on first have to be computed. This is a recursive process along the dependency graph, which only makes sense if the library does not contain cycles. After that, library analysis is started which guarantees local consistency between the analysed component and the components it depends on. The resulting context relation of a component that depends on other components is computed as follows:

$$cr[\![t :: t_1, ...t_n]\!] = basicrelation[\![t]\!] \sqcap \bigsqcap_{i=1}^{n} cr[\![t_i]\!],$$
$$cr[\![T]\!] = basicrelation[\![T]\!],$$
$$cr[\![\ \#Include\ Id(parameters)]\!] =$$
$$instantiate(signature\_scheme[\![Id]\!], parameters)$$

As mentioned above, there exist other, language-specific dependencies; these are not mirrored in the above formulas.

For imported modules, their signature is directly used as a basic context relation. For implementation modules, the definition module's signature is fed into the analysis, but in contrast to the first case, "opaque" types in the definition modules must be refineable, which causes some extra trouble. Also, we would like to have incremental analysis after a change to a component. These and other Modula-2–specific topics are not very relevant for the purpose of this paper, and hence left out.

4 Two reusable components

We now will show a complete example of a very small Modula-2 library consisting of two reuseable polymorphic components, namely a fixed size stack called "boundedStack" and an iterative "quicksort" for arrays containing arbitrary elements. The "quicksort" uses "boundedStack" for bookkeeping of partitions which still have to be sorted. "quicksort" itself is used by two programs namely "sortInt.prg" and "sortTree.prg" which sort a list of integers and an array of trees respectively.

First we show how the polymorphic components "boundedStack.comp" and "quicksort.comp" are developed and how the analysis algorithm supports the development of such library components.

The analysis of "boundedStack", whose program text was shown in section 3.1, results in the following interface, as displayed by NORA. Two warnings are produced, which signalize that "elemType" and "stackSize" are not defined within the component – they are the source of polymorphism, and hence are implicit parameters of the component.

```
Interface for component "boundedStack.comp"

In line 4:
  WARNING: Undefined Identifier "elemType"
  WARNING: Undefined Identifier "stackSize"
elemType:  CLASS = TYPE
           TYPE  = <non-structured type>
stackSize: CLASS = <expression>
           TYPE  = INTEGER or CARDINAL
push:      CLASS = PROCEDURE
           TYPE  =  (VALUE PARAMETER:
                        <non-structured type>): _
pop:       CLASS = PROCEDURE
           TYPE  =  (): <non-structured type>
initStack: CLASS = PROCEDURE
           TYPE  =  (): _
isFull:    CLASS = PROCEDURE
           TYPE  =  (): BOOLEAN
isEmpty:   CLASS = PROCEDURE
           TYPE  =  (): BOOLEAN
```

Analysis has inferred that "stacksize" must be an expression of type "INTEGER or CARDINAL". Furthermore, the analysis reveals that "boundedStack" does not have unlimited polymorphism, since – according to the context conditions of Modula-2 – "elemType" must be a non–structured type. This demonstrates how NORA can inform the programmer about the possible usage contexts of his components. (Changing "pop" to a procedure returning the popped result as an output parameter would allow "elemType" to be instantiated with any type).

The polymorphic quicksort component uses and instantiates the library component "boundedStack". In order to avoid name clashes between the stack and the quicksort component, explicit parameters are used in the "#include" statement. In particular, both the stack and quicksort use "elemtype", hence the #include explicitely states that CARDINALs should be pushed onto the stack — if one would rely on the default rule that parameters are bound by the usage context, a type conflict would result. The source code looks as follows (this program is used for illustration purposes only; a more realistic quicksort would push only the indices of the larger partition):

```
PROCEDURE quicksort(VAR a: ARRAY  OF elemType);
  VAR
    pivot: elemType;
    pivotindex, left, right: CARDINAL;
    k: CARDINAL;
  CONST
    maxStack = 100;

#include boundedStack.comp (stackSize = maxStack,
                            elemType = CARDINAL)

  PROCEDURE findpivot(a: ARRAY  OF elemType;
                      i, j: CARDINAL): CARDINAL;
    ...
  END findpivot;

  PROCEDURE partition(VAR a: ARRAY  OF elemType;
                      i, j: CARDINAL;
                      pivot: elemType): CARDINAL;
    ...
      WHILE greater(a[j], pivot) DO
        ...
    ...
  END partition;

BEGIN
  left := 0; right := HIGH(a);
  initStack(); push(left); push(right);
  pivotindex := 0;
  REPEAT
    IF pivotindex = 0 THEN
      right := pop(); left := pop()
    END;
    pivotindex := findpivot(a, left, right);
    IF pivotindex <> 0 THEN
      pivot := a[pivotindex];
      k := partition(a, left, right, pivot);
      push(k); push(right); right := k - 1
    END
  UNTIL isEmpty()
END quicksort;
```

The analysis computes the following interface for "quicksort". Again the undeclared identifiers "elemType" and "greater" are the source of polymorphism. NORA checks that "boundedStack" is correctly used and instantiated within "quicksort", but "boundedStack" does not influence the "quicksort" interface.

```
Interface for component "quicksort.comp"

In line 2:
  WARNING: Undefined Identifier "elemType"
In line 24:
  WARNING: Undefined Identifier "greater"
elemType:  CLASS = TYPE
           TYPE  = type
quicksort: CLASS = PROCEDURE
           TYPE  =  (REF PARAMETER:
                       ARRAY [<ordinal type>]
                       OF <type>): _
greater:   CLASS = PROCEDURE Object
           TYPE  =  (<parameter description>:
                       <type>;
                     <parameter description>:
                       <type>): BOOLEAN
arraySize: CLASS = <expression>
           TYPE  = INTEGER or CARDINAL
```

Our library by now consists of two reuseable polymorphic components, where one already uses the other. Next we will show how "quicksort" can be used in two completely different contexts. The first example is an application of "quicksort" to a list of integers, in the second example an array of trees will be sorted.

```
MODULE sortInt;
  TYPE
    elemType = INTEGER;
  VAR
    a: ARRAY [1 .. 117] OF elemType;
  PROCEDURE greater(VAR res: BOOLEAN;
                    x, y: elemType);
  BEGIN
    res := x > y;
  END greater;

#include quicksort.comp

BEGIN ...
  quicksort(a); ...
END sortInt.
```

The program module "sortInt" uses "quicksort" in order to sort an array of integers. The interface shows that "quicksort" is appropriatly instantiated. Note that the default parameter mechanism for "#include" is appropriate here, since accidentially (!) the free identifiers of the quicksort component are correctly bound at the usage site. NORA detects an error anyhow: "quicksort" expects a boolean function comparing two elements, but "greater" is defined as a procedure with a boolean reference parameter returning the result.

```
Interface for component "sortInt.prg"

In line 24:
  WARNING: Undefined Identifier "print"
In line 22:
  WARNING: Undefined Identifier "fill"
In line 19
  ERROR: "greater" has incompatible attributes
  expected attribute:
     CLASS = PROCEDURE
     TYPE  =  (REF PARAMETER: BOOLEAN;
                VALUE PARAMETER: INTEGER;
                VALUE PARAMETER: INTEGER): _
  actual attribute:
     CLASS = PROCEDURE Object
     TYPE  =  (<parameter description>: <type>;
                <parameter description>: <type>):
               BOOLEAN
  Different number of parameters
elemType:  CLASS = TYPE
           TYPE  = INTEGER
greater:   CLASS = PROCEDURE
           TYPE  =  (REF PARAMETER: BOOLEAN;
                      VALUE PARAMETER: INTEGER;
                      VALUE PARAMETER: INTEGER): _
arraySize: CLASS = CONSTANT
           TYPE  = INTEGER or CARDINAL
quicksort: CLASS = PROCEDURE
           TYPE  =  (REF PARAMETER:
                      ARRAY [INTEGER or CARDINAL]
                        OF INTEGER): _
```

The second program module uses "qicksort" for sorting a forest. The user did not yet decide about the implementation of trees, hence the type "treeElem" is still missing. The missing program text is represented by a placeholder (printed as a comment), which is a standard technique in language-based editors. Furthermore, "greater" is only partially defined.

```
MODULE sorttrees;
  TYPE
    elemType = POINTER TO treeElem;
    treeElem = (*type*);
  VAR
    forest: ARRAY [0 .. 44] OF elemType;

  PROCEDURE greater(tree1, tree2: elemType): BOOLEAN;
    PROCEDURE findlargestkey(tree: elemType): CARDINAL;
      ...
    END findlargestkey;
  BEGIN
    RETURN
     findlargestkey(tree1) > findlargestkey(tree2);
  END greater;

#include quicksort.comp

BEGIN
  ...
  quicksort(forest);
  ...
END sorttrees.
```

Since the representation of trees is still undefined, "quicksort" gets only partially instantiated, as shown in the interface.

```
...
elemType:  CLASS = TYPE
           TYPE  = POINTER TO <type>
greater:   CLASS = PROCEDURE
           TYPE  =  (VALUE PARAMETER:
                        POINTER TO <type>;
                     VALUE PARAMETER:
                        POINTER TO <type>): BOOLEAN
arraySize: CLASS = CONSTANT
           TYPE  = INTEGER or CARDINAL
quicksort: CLASS = PROCEDURE
           TYPE  =  (REF PARAMETER:
                        ARRAY [INTEGER or CARDINAL]
                        OF POINTER TO <type>): _
```

This last example demonstrates that the analysis also works for partially incomplete components.

5 Implementation issues

The original inference engine for relational analysis was implemented in Pascal as part of the PSG system [1]. It was very sophisticated and could analyse 1000 lines of Modula-2 in less than 5 seconds on a SUN 3/60. In the scope of the PSG project, relational definitions of context conditions were developed not only for Modula-2, but also for Pascal, ADA [7], Fortran77 [16], and a number of experimental languages.

Unfortunately, this implementation was unable to analyse a whole library, it could only analyse isolated components. Furthermore, it suffered from increasing unintelligibility, since the implementation evolved over a period of seven years. We therefore decided to reimplement the inference engine in the functional language SAMPλE. The new implementation now handles complete libraries and is integrated into NORA. But it is an order of magnitude slower than the original system, and the maximum component size is limited. In order to make our approach usable for large-scale libraries, another reimplementation is perhaps necessary.

6 Future work

Component retrieval: Polymorphic components as described in this paper are the basis for unification-based component retrieval. NORA's retrieval subsystem allows to search components by type schemes or signature schemes. The user can for example click at an identifier and find out whether there are objects in the library with equal or similar type characteristics. The method is based on order-sorted unification and AC1–unification of signature schemes. It is currently under implementation, and we hope to be able to report in the near future.

Fully hygienic components: Our parameterized components are only 50% hygienic in the sense of Kohlbecker. Although component parameters prevent unwanted bindings of free component variables at usage site, it may still happen that actual parameters are erroneously bound to local variables inside the component. Hence, in rare cases incorrect component uses can go undetected (and will show up later when the program is actually compiled). We plan to incorporate Clinger's algorithm in order to solve this problem.

Arbitrary program fragments as component parameters: At the moment it is not possible to use arbitrary program text as actual component parameter. This extension could however be very useful, provided that actual parameters constitute syntactic units (e.g. expressions). It implies that parameter substitution can no longer be done by changing some bindings in the component's interface; instead, actual parameters must be syntactically and semantically analysed, and consistency of the component with parameters substituted must be checked.

Configuration management: We expect that NORA's configuration management is improved by the techniques described in this paper: in addition to consistency of a configuration thread as specified by the configuration rules, a particular thread must of course be consistent with respect to component interfaces as well. Hence, NORA can detect that certain configuration threads are impossible due to interface constraints. Configuration management and semantic analysis are however not yet integrated.

Polymorphic components for polymorphic languages: What happens if we want to apply our concept to a language which already employs polymorphism, such as ML? It has been shown in [14] that relational analysis will compute correct typings for such languages. But unfortunately, type inference interferes with the component parameter mechanism: imagine that an actual component parameter is itself a polymorphic object. Correct typing then requires to allow different instantiation of the type of this actual parameter. But inside the component, type inference treats component parameters as monomorphic, unifying all locally derived types. This is incorrect, but in accordance with the Damas-Milner type system, where Lambda-bound variables cannot be polymorphic. It is a long–standing open problem whether a decideable type system can be devised which allows Lambda-bound variables to be polymorphic [2].

7 Conclusion

We have shown how polymorphic type inference, as introduced by Milner, can be utilized in "ordinary" monomorphic programming languages, in order to improve component reuse. Our approach gives the programmer more type safety and better retrieval possibilities. In particular, it

- guarantees library consistency without the need to reanalyse all uses of components
- provides signature schemes, which tell the programmer the possible usage contexts of a component
- allows to retrieve components by signature unification.

However, it does not avoid the need for object code duplication – NORA components can be preanalysed, but not precompiled as in "real" polymorphic languages. This slight, but inevitable disadvantage of our approach is compensated by the observation that polymorphic languages need more expensive runtime environments.

Acknowledgements. Matthias Kievernagel implemented the name resolution, and Andreas Zeller implemented the NORA kernel. Bernd Fischer contributed valuable discussions.

The work described in this paper is funded by the Deutsche Forschungsgemeinschaft, grant He1170/4–1.

8 References

[1] Bahlke, R., Snelting, G.: The PSG System: From Formal Language Definitions to Inter-active Programming Environments. ACM TOPLAS 8, 4 (October 1986), pp. 547 – 576.

[2] Barendregt, H., Hemerik, K.: Types in Lambda Calculi and Programming Languages, Proc. 3rd European Symposium on Programming, LNCS 432, pp. 1 – 35.

[3] Beidler, J., Jackowitz, P.: Consistent Generics in Modula-2. ACM SIGPLAN Notices 21, 4 (April 1986), pp. 32 – 41.

[4] Clinger, W., Rees, J.: Macros That Work. Proc. 18th Principles of Programming Languages, ACM 1991, pp. 155 – 162.

[5] Czyzowics, J., Iglewski, M.: Implementing Generic Types in Modula-2. ACM SIGPLAN Notices 20, 12 (December 1985), pp. 26 – 32.

[6] Damas, L., Milner, R.: Principal Type Schemes for Functional Programs. Proc. 9th Principles of Programming Languages, ACM 1982, pp. 207 – 217.

[7] Grosch, F.-J., Snelting, G.: Inference-Based Overloading Resolution for ADA. Proc. Programming Language Implementation and Logic Programming, Linkøping 1990, LNCS 456, pp. 30 – 44.

[8] Kohlbecker, E., Friedman, D., Felleisen, M., Duba, B.: Hygienic Macro Expansion. Proc. Lisp and Functional Programming, ACM 1986, pp. 151 – 159.

[9] Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17, 3 (1978), pp. 348 – 375.

[10] Rittri, M.: Using Types as Search Keys in Function Libraries. Proc. Functional Languages and Computer Architecture, ACM 1989, pp. 174 – 183.

[11] Rittri, M.: Retrieving Library Identifiers via Equational Matching of Types. Proc. 10th Conference on Automated Deduction, LNCS 449, pp. 603 – 617.

[12] Rollins, E., Wing, J.: Specifications as Search Keys for Software Libraries. Proc. International Conference on Logic Programming, Paris 1991.

[13] Runciman, C., Toyn, I.: Retrieving Re-Usable Software Components by Polymorphic Type. Proc. Functional Languages and Computer Architecture, ACM 1989, pp. 166 – 173.

[14] Snelting, G.: The Calculus of Context Relations. Acta Informatica Vol. 28 (Mai 1991), pp. 411 – 445.

[15] Snelting, G., Grosch, F.-J., Schroeder, U.: Inference-Based Support for Programming in the Large. Proc. 3rd European Software Engineering Conference, Milano 1991. Springer Verlag, LNCS 550, pp. 396 – 408.

[16] Snelting, G., Thies, C.: Programming Tools for the Suprenum Supercomputer. Proc. 3rd International Workshop on Software Engineering & its Applications, Toulouse 1990, pp. 951 – 964.

[17] Sugiyama, K., Tagawa, S., Toda, M.: Methods for Visual Understanding of Hierarchical System Structures. IEEE Transaction on Systems, Man and Cybernetics 11, 2 (1981), S. 109 – 125.

[18] Wiener, R., Sincovec, R.: Two Approaches to Implementing Generic Data Structures in Modula-2. ACM SIGPLAN Notices 20, 12 (June 1985).