OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Type Checking and Parameterized Types

David Sandberg
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

85-1-1

Type Checking and Parameterized Types

David Sandberg
Oregon State University

A method for implementing parameterized types is given. Two simplifying restrictions are assumed: types are only parameterized with other types, and assignment is like that of SNOBOL, CLU, and Smalltalk. An algorithm for type checking that handles parameterized types and overloading is presented.

1. Introduction

The idea of parameterized types or "generics" is more than a decade old, but has been used in only a few expermental languages. Although Ada[3] includes generics, Ada does not have true parameterized types. One reason for this is that the implementation of parameterized types is not well understood. This paper shows how parameterized types can be restricted so that they have a straightforward implementation. The majority of this paper describes an algorithm for type checking that handles parameterized types and overloading.

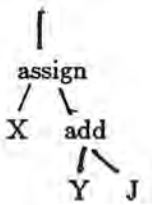
Two simplifying restrictions are assumed. The first is that types are parameterized only with other types. This does not allow the size of an array to be a parameter, which requires arrays to be defined differently than they are in the Algol family of languages. Removing the size of an array from the type can be considered an advantage since the proper size for an array is usually not known at compile time anyway. The author's experience with parameterized types has not shown much need for types parameterized with anything but another type. Experience with CLU seems to indicate the same thing[2].

The second simplifying restriction is that every object can be represented in one word. If the object is bigger than one word, a pointer to the object is stored. Also, an assignment statement will just copy the pointer to the object and not the whole object. This is the kind of assignment that is used in SNOBOL[6], CLU[8], and Smalltalk[4]. Treating assignment this way makes the implementation of assignment and parameter passing independent of the types of the objects. This makes code generation a great deal simpler. Rosenberg[10] has looked at implementing generics without this second restriction.

After making these restrictions the most complex part of dealing with parameterized types is the type checking. The next sections describe the type checking process.

2. Informal Description of Type Checking.

To illustrate how the type check works let us look at the follow simple statment: X := Y+J. By removing the infix notation the following equivalent statement is formed: assign(X,add(Y,J)). This in turn is a linear representation of a tree. The function name is the root of the tree and its arguments are the subtrees of the root.



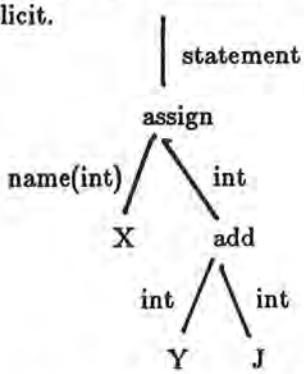
Standard compiler technology allows the program text to be easily turned into such a tree. An arc into the root of the tree is add for use in the following description.

The symbol table is a finite set of symbols. Each symbol is a pair consisting of a name and a type. The type may be simple like *int* or be parameterized like *proc* (*int* ,*int* ,*int*). The type *proc* will be the type of a procedure. The last parameter of *proc* will be the result type of the procedure. If the result type is *statement* , then no value is returned. The other parameters of *proc* will be the types of the arguments. A parameterized type *name* is used to distinguish between l-values and r-values. The l-value of a variable of type T will be represented by *name* (T). The r-value will be represented by T . The symbols used in the above example are:

(*assign* ,*proc* (*name* (*int*),*int* ,*statement*))
(*add* ,*proc* (*int* ,*int* ,*int*)))
(X ,*proc* (*name* (*int*)))
(Y ,*proc* (*name* (*int*)))
(J ,*proc* (*name* (*int*)))

The variables are typed as procedures. This makes the following description more uniform. In practice the type of X would be *name* (*int*) and the algorithm modified slightly to handle this.

The process of type checking will label each node of the tree with a symbol and label each arc of the tree with a type. The label on the arc entering the root is labeled with the expected type for the whole tree. A node can be labeled with a element of the symbol table, (k ,*proc* (t_1,t_2,t_3, . . . , t_n)), if k is the name of the tree node, the number of sons of the node is $n-1$, the label on the arc of the i'th sub tree is t_i , and the label on the father arc l_0 is such that $l_0=t_n$ or *name* (l_0)=t_n . Permitting t_n to equal either l_0 or *name* (l_0) makes one level of dereferencing implicit.



Overloading is done by having more than one symbol in the symbol table with the same name. For example:

(*assign* ,*proc* (*name* (*int*),*int* ,*statement*))
(*assign* ,*proc* (*name* (*real*),*real* ,*statement*))

If more than one labeling of the tree exists, there is an ambiguity in the semantics. If no labelings exist then there is a type conflict or an identifier is not declared.

Instead of overloading the assignment operator, it would be better to have one rule to handle all cases. This can be done by introducing *free types*. Let α_1 represent a free type in the following symbol:

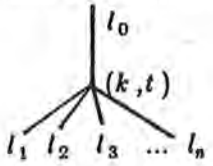
$(assign, proc\ (name\ (\alpha_1), \alpha_1, statement))$

When this symbol is used to label a node of the tree, a type is first chosen and substituted for α_1 in the type of the symbol. A different type may be chosen for each node of the tree. The rest of the matching takes place as without free types.

3. Formal Description

This section gives a more formal description of the type checking process. Let I be a set of symbols that represent identifiers. We will assume *proc* and *name* are in I. Let $\alpha_0, \alpha_1, \alpha_2, \cdots$ be a set of symbols, not in I, used to represents free types. Let the set of types T be the smallest set such that $I \subset T$ and if $t_i \in T$ and $c \in I$ then $c\,(t_1, t_2, t_3, \ldots, t_n) \in T$. Let T' be the smallest set such that $I \subset T'$, $\alpha_i \in T'$, and if $t_i \in T'$ and $c \in I$ then $c\,(t_1, t_2, t_3, \ldots, t_n) \in T'$. The difference between T and T' is that T' allows free types in the types whereas T does not. A symbol table will be a finite subset of Σ where Σ is the set of ordered pairs whose first component is a member of I and the second component a member of T' of the form $proc\,(t_1, t_2, \ldots, t_n)$ where $n \geq 1$. We denote a *substitution* by a finite sequence $\beta = t_0, t_1, \ldots, t_n$ where $t_i \in T'$. For any $q \in T'$, $q \mid \beta$ will denote the element of T' formed by simultaneously replacing each occurrence of α_i by $t_i, 0 \leq i \leq n$ in q. A substitution β is a *unifier* for q and q' if $q \mid \beta = q' \mid \beta$. β is a *most general* unifier for q and q' if for every unifier β' of q and q' there exists a substitution γ such that $(q \mid \beta) \mid \gamma = q \mid \beta'$. The term unifier comes from theorem proving where algorithms for finding most general unifiers can be found[9].

The type checking problem is: given a set of symbols $S \subset \Sigma$, the expected type $r \in T$, and an ordered tree whose nodes are labeled with elements of I, produce a labeling of the nodes and arcs of the tree such that the node labels are elements of S, the arc labels are elements of T, the arc into the root is labeled with r and for each node of the tree



where $l_0, l_1, l_2, \ldots, l_n$ are arc labels, k must be the original label of the node, and there must exist a substitution d such that $t \mid d = proc\,(l_1, l_2, \ldots, l_n, l_0)$ or $t \mid d = proc\,(l_1, l_2, \ldots, l_n, name\,(l_0))$. (In practice we also require that the choice of the arc labels and node labels to be unique.)

4. The Algorithm

We now give an algorithm for finding such a labeling. Two passes over the tree will be made. In the first pass we work from the leaves to the root of the tree labeling each arc with a

subset of T' . This set encodes all potential labelings. Suppose we have already labeled all arcs with the sets $L_1, \ldots, L_n$ to the sons of a tree node originally labeled k . Furthermore we will assume the sets of free symbols used in describing, $S, L_1, \ldots, L_n$ are all disjoint. If they are not disjoint, a renaming of the free symbols will make them disjoint. We will construct a set L_0 to label the arc entering this node. Let D be a subset of S containing all the elements of S of the form $(k, proc(t_1, t_2, \ldots, t_{n+1}))$. D contains all the symbols in S with the correct name and right number of sons for labeling the current node. For each element of D and for each possible choice of $l_1, l_2, \ldots, l_n$ such that $l_i \in L_i$, if there exists a most general unifier d for $proc(l_1, l_2, \ldots, l_n, t_{n+1})$ and $proc(t_1, t_2, \ldots, t_{n+1})$, then add $t_{n+1} \mid d$ to L_0. If $t_{n+1} \mid d$ is of the form $name(t')$, then also add t' .

After the first pass over the tree is complete there should exist a substitution d such that for some t in the set labeling the arc into the root $t \mid d = r$ where r is the expected type. If there is no such d, then no labeling is possible.

The second pass will label each node with an element of S and reduce each set of labels on the arcs to a single label. First the arc into the root is labeled with the expected type r. If the arc into a node already has been labeled in the second pass with l_0 then, in the set D constructed in the first pass, find an element $(k, proc(t_1, \ldots, t_n))$ and $l_1, \ldots, l_{n+1}$ such that 1) $l_i \in T$, and 2) there is a unifier for l_i and an element of L_i, and 3) there is a unifier for $proc(t_1, \ldots, t_{n-1}, t_n)$ and $proc(l_1, \ldots, l_{n-1}, l_0)$ or $proc(l_1, \ldots, l_{n-1}, name(l_0))$. If there is more than one way to choose the element of D and the l_i, then there is more than one labeling of the tree. Label the node with $(k, proc(t_1, \cdots t_{n+1}))$ and the arc to the i'th son with l_i.

In this algorithm the sets that label the arcs can grow to size 2^{l-1} where l is the number of nodes in the tree. If S is $\{(a, proc(z)), (b, proc(\alpha_1, x(\alpha_1))), (b, proc(\alpha_1, y(\alpha_1)))\}$ and the trees that have only one leaf label with a and $l-1$ internal nodes labeled with b are used, then such behavior will be achieved.

The type matching problem is NP-complete[1]. Any arbitrary labeling can be checked to see whether it is a solution in polynominal time. Thus the problem is in NP. To show that it is NP-complete we will transform satisfiability of CNF boolean expressions. Give each disjunctive clause in the expression a name, say, $a_1, \ldots, a_n$. Let k be the number of distinct variables used in the whole expression. Let $x_1, \ldots, x_k$ be their names. For each clause a_i add a set of elements to S . If x_j appears in the clause add

$$(a_j, proc(z(\alpha_1, \ldots, \alpha_{j-1}, t, \alpha_{j+1}, \ldots, \alpha_k), z(\alpha_1, \ldots, \alpha_{j-1}, t, \alpha_{j+1}, \ldots, \alpha_k)))$$

If $\bar{x}_j$ appears in the clause add

$$(a_j, proc(z(\alpha_1, \ldots, \alpha_{j-1}, f, \alpha_{j+1}, \ldots, \alpha_k), z(\alpha_1, \ldots, \alpha_{j-1}, f, \alpha_{j+1}, \ldots, \alpha_k)))$$

Also add $(b, proc(z(\alpha_1, \ldots, \alpha_k)))$ and $(c, proc(z(\alpha_1, \ldots, \alpha_k), t))$. Now if we use t as the expected symbol and a tree with one leaf that has the root label c, the leaf labeled b, and the other nodes label with $a_1, \ldots, a_n$ then there will be a labeling for this tree if and only if the boolean expression is satisfiable.

5. Code Generation

After the type checking is done, code generation is fairly straightforward. If no operations besides assignment are required on the free types in a procedure, then the code generation is no different than in a conventional compiler. In CLU any addition operations on the free types must be specified in a **where** clause. For example,

```
equal=proc(lst1,lst2:list[t]) returns (bool)
  where t has equal: proctype(t,t) returns (bool)
```

(To obtain the strict CLU text replace "list[t]" by "cvt".) This specifies that a equality procedure is needed on type parameter "t". Alphard has an **assumes** clause that serves the same purpose. The operations required on the free types should be specified and not left for the compiler to determine. These specifications are needed for verification, for maintenance, and for reuse of the procedure.

The simplest way to generate code for procedures with other operations specified is to have the compiler pass the operations as additional parameters to the procedure. An optimizing compiler could generate separate procedures for each distinction set of required operations in some cases, but not in all cases because the number of distinct sets could be infinite[5].

6. Conclusion

The techniques for implementing parameterized types described above have successfully been used in a compiler for an experimental language, X2. In practice, the NP-completeness of the algorithm has not been a problem because the size of the sets used to label the arc is one or two elements 99% of the time. The algorithm was modified to label the nodes with other types beside *proc* . For example, a type *star(sontype,fathertype)* was defined that will match a node with any number of sons that all have the type *sontype* and return a type *fathertype*.

It seems unlikely that either of the simplifying restrictions can be removed without greatly complicating the implementation of parameterized types. If this is so, then it is unlikely that parameterized types will come into widespread use in the near future, as this would imply a shift from languages like Fortan, Pascal[7], Modula-2[13], and Ada to languages more like CLU, SNOBOL, and Smalltalk.

7. REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] R. Atkinson, B. Liskov, and R. Scheifler. Aspects of Implementing CLU. *Proc. 1978 ACM Annual Conference,* Washington, D.C.,123-129.

[3] *Ada Programming Language.* Department of Defense, Military Standard MIL-STD-1815A, January, 1983.

[4] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[5] D. Gries and N. Gehani. Some Ideas on Data Types in High-Level Languages. *CACM* 20(6), June 1977, 414-420.

[6] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language,* 2nd ed. Prentice-Hall, Englewood Cliffs, 1971.

[7] K. Jensen, and N. Wirth. *Pascal User Manual and Report,* 2nd ed. Springer-Verlag, New York, 1974.

[8] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *CACM* 20(8), August 1977, 564-576.

[9] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

[10] J. Rosenberg. *Generating Compact Code for Generic Subprograms.* Thesis, Carnegie-Mellon University, August 1983. Published as CMU Tech. Rep. CMU-CS-83-150.

[11] M. Shaw, ed. *ALPHARD: Form and Content.* Springer-Verlag, New York, 1981.

[12] B. Wegbreit. The Treatment of Data Types in EL1. *CACM* 17(5), May 1974, 251-264.

[13] N. Wirth. *Programming in Modula-2,* Springer-Verlag, New York, 1983.