Inferring Recursive Structures in Types in Prolog Programs using Abstract Interpretation

Fumiaki Kamiya

UCSC-CRL-96-17
July 25, 1996

Baskin Center for
Computer Engineering & Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

One way to prove termination of a logic program is to show that input terms passed to each recursive procedure decrease in size between successive calls to the same procedure with respect to some appropriately defined norms. Except when all the input terms are ground, finding the appropriate norms is generally not easy. Barring the use of numbers and arithmetics, recursive calls are controlled by recursively constructed terms. Thus, one way to automatically find the appropriate norms would be to automatically identify recursive structures in term types. This report describes a work in progress that attempts to infer such recursive structures using abstract interpretation.

Keywords: Termination analysis, abstract interpretation, recursive structures in term types.

1 Introduction

Termination analysis of logic programs has attracted a lot of interest within the logic programming community. For a survey of relevant works, see [2]. One way to prove termination is to show that input terms passed to each recursive procedure decrease in size between successive calls to the same procedure with respect to some appropriately defined norms. With the exception when input terms are all ground, finding the appropriate norms is generally not easy. However, in order to automate the process of termination analysis, a mechanism to automatically find these norms is necessary.

Barring numbers and arithmetics to control recursive calls, all recursive calls are controlled by recursively constructed terms. Thus, one way to automatically find the appropriate norms would be to automatically identify recursive structures in term types.

This report describes a work in progress that attempts to infer recursive structures in term types. The method is based on abstract interpretation ([1]); the program is converted into a program on the abstract domain of types and then evaluated in a bottom-up fashion in the style of forward reasoning.

2 Recursive Calls and Recursive Terms

Barring numbers and arithmetics (and other non-logical constructs) to control recursive calls, the only way to control recursive calls is to use recursively constructed terms. Thus, one way to automatically find the appropriate norms would be to automatically identify recursive structures in terms permissible as arguments for each procedure. Once these are identified, we can test for termination by checking that any goal has its input terms' recursive structures instantiated so that the terms can safely control recursive calls.

Example. Consider the following program:

```
len([], 0).
len([_|T], s(S)) :- len(T, S).
```

The recursive structure in the first argument of `len` is in the second argument, if at all, and the recursive structure in the second argument of `len` is in the first argument, if at all. Because of this, it is obvious that although goals

```
?- len([_|[_|[]]], _).
```

and

```
?- len(_, s(s(0))).
```

will terminate,

```
?- len([[[]|_]_], _).
```

will not. □

3 Types, Type Expressions, and Type Labels

Types are chosen to be the abstract domain in our application of abstract interpretation. Formally, a type is defined to be the set of values variables and constants of the type may take. However, our interest is not in trying to infer all possible values of the type but rather to infer a small number of construction rules that could generate all possible values. Further, it is not necessary that the set of construction rules be "exact". We are interested in knowing the presence of any recursive structures in types; therefore, some amount of redundancy in the construction rules is acceptable, particularly if this makes the construction rules simpler.

Example. Consider the following program:

```
even(0).
even(s(s(N))) :- even(N).
```

Considering `0` to be the numerical zero and `s` to be the successor function, `even(`X`)` succeeds if X is a non-negative even integer. The type of X can be enumerated as:

$\{$`0`, `s(s(0))`, `s(s(s(s(0))))`, ...$\}$

An exact set of construction rules for this type is

typeof(X) ::= `0`

typeof(X) ::= `s(`T`)`

However, for our purpose, the following set of construction rules suffices.

typeof(X) ::= `0`

typeof(X) ::= `s(`T`)`

□

Type expressions are expressions involving types and describe other types. Let c be a constant, f be an n-ary functor, and $T_1, \ldots, T_n$ be n types. Then c denotes the type whose only value is c, and $f(T_1, \ldots, T_n)$ denotes the type whose values have the form f`(`x_1`, ...,`x_n`)` where $x_1 \in T_1, \ldots, x_n \in T_n$.

Type labels are names introduced into the abstracted program for the purpose of referring to "key" types. Unique type labels are generated and assigned to each argument position of each procedure. The reason for separate type labels for different clauses of the same procedure is so that type inference can proceed with respect to each clause as opposed to the set of clauses that (completely) define the procedure. In a way, the type inference scheme described in this report can be thought of as trying to describe these type labels. Note that in the abstracted program, we can introduce the type labels using assertions; this is because the domain of the abstracted programs is types.

Example. In this report, type labels are generated systematically using the following template:

`typelabel$`*clause*`.`*arg*

where *clause* is the position of the clause in the program and *arg* is the argument position. The first clause is at position 1 and the first argument is at position 1.

Continuing with the earlier example using `len`, we generate four type labels—`typelabel$1.1`, `typelabel$1.2`, `typelabel$2.1`, and `typelabel$2.2`—and add them to the abstracted program as assertions.

```
len(typelabel$1.1, typelabel$1.2).
len(typelabel$2.1, typelabel$2.2).
```

□

4 Approach

We consider normalized programs. Normalized programs can be obtained from non-normalized programs by a straightforward syntactical transformation. The reason for the use of normalized programs is so that all unifications become explicit.

Example. Consider again the following program:

```
len([], 0).
len([_|L], s(S)) :- len(L, S).
```

A normalized version of the above program would be:

```
len(A01, A02) :- A01 = [], A02 = 0.
len(A01, A02) :- A01 = [_|A012], A012 = L, A02 = s(A021), A021 = S,
                 len(A11, A12), A11 = L, A12 = S.
```

□

At this time, we can infer some information about the types in the head of each clause by looking at the body of the clause.

Example. Continuing with the example, we can infer from the first clause that one successful argument for `len` is `([], 0)`. Also, from the second clause, we can infer that another successful argument for `len` has a term whose top functor is `.` (the list constructor) as its first argument and a term whose top functor is `s` as its second argument. □

Note that in the previous example, type inference was essentially done by comparing the left and right hand sides of `=`s. Although similar in form, the information obtained from the unifications depended on what were on the two sides. For instance, the `=` in the first clause meant that the constant term on the right hand side was the only instance of the type on the left hand side, whereas in the second clause, the `=` meant that the type appearing on the left hand side has the form appearing on the right hand side (e.g., `A01 = [_|A012]`), or that the two types are the same (e.g., `A012 = L`)[1].

Let us now formalize this by introducing the rules as axioms for type inference.

- If the goal is X `=` y where X is a variable and c is a constant, then X is a singleton type whose only element is c.
- If the goal is X `=` f`(`Y_1`,...,`Y_n`)` where X and $Y_1, \ldots, Y_n$ are variables for some $n \geq 1$, then the type of X is $f(Y_1, \ldots, Y_n)$.
- If the goal is X `=` Y where X and Y are two variables, then the types of X and Y are the same.

To disambiguate the overloaded use of `=`, we convert the normalized program so that each use of `=` is rewritten using the appropriate predicate—either `instance` or `iff`. `instance(`X`,` c`)` means that c is the only instance of type X and is used for the first case above. `iff(`X`,` Y`)` means that X and Y have the same type and is used for the second and third cases above.

Example. Continuing with the example, we obtain the following code after the transformation.

```
len(A01, A02) :- instance(A01, []), instance(A02, 0).
len(A01, A02) :- iff(A01, [_|A012]), iff(A012, L),
                 iff(A02, s(A021)), iff(A021, S),
                 len(A11, A12), iff(A11, L), iff(A12, S).
```

□

To be able to infer recursive structures in types, we now introduce type labels described in Section 3.

Example. Continuing with the example, the complete abstracted program, with the type labels, looks like this:

[1] As I write this report now, I feel that the distinction was not necessary. This may be changed in the future.

```
len(A01, A02) :- instance(A01, []), instance(A02, 0).
len(A01, A02) :- iff(A01, [_|A012]), iff(A012, L),
                 iff(A02, s(A021)), iff(A021, S),
                 len(A11, A12), iff(A11, L), iff(A12, S).
len(typelabel$1.1, typelabel$1.2).
len(typelabel$2.1, typelabel$2.2).
```

□

Now, using bottom-up evaluation on the abstracted program, we can infer recursive structures in types.

Example. Continuing with the example, a straightforward bottom-up evaluation using the second and the fourth clauses yields that a success pattern for `len` involving the second clause as the top clause is `([_|typelabel$2.1], s(typelabel$2.2))`. Since `typelabel$2.1` and `typelabel$2.2` are the type labels for the head of this clause, we have inferred that

`typelabel$2.1` ::= [?|`typelabel$2.1`]

`typelabel$2.2` ::= *s*(`typelabel$2.2`)

where '?' is used to mean "don't care". These tell us that recursive structures are present in both arguments (because, in each inferred type relation, the type labels on the left hand side also appears on the right hand side), that for the first argument position of `len`, the recursive structure is present in the second argument position of . (because that's where the type label `typelabel$2.1` occurs on the right hand side), and that for the second argument position of `len`, the recursive structure is present in the first argument position of `s`. □

5 Some Implementation Details

During the evaluation of the axioms, relationships between different (type) variables and membership of constants to types are obtained. These information are passed from one goal to another during bottom-up evaluation of the abstracted program via variables added by the translator. This is done so that no special feature is required in the bottom-up evaluator.

For each clause in the program, the head and the goals in the body are added two more parameters. Both parameters precede original parameters in the program. The design choice of these parameters preceding original parameters was made so that these parameters will always appear at the same argument positions.

Example. Continuing with the example, `len` is increase of its number of arguments to four. Each goal will now look like this:

```
len(TypeRelIn, TypeRelOut, List, Len)
```

The meaning of the arguments are:

`TypeRelIn`	List of type relations known before the evaluation of this goal.
`TypeRelOut`	List of type relations known after the evaluation of this goal.
`List`	Same as the first argument of the original `len`.
`Len`	Same as the second argument of the original `len`.

□

The head as well as the goals in the body will share variables so that the type relations obtained during bottom-up evaluation will flow through the clause.

Example. Continuing with the example, the program now looks like this:

```
len(TypeRelIn, TypeRelOut, A01, A02) :- instance(TypeRelIn, TypeRel1, A01, []),
                                        instance(TypeRel1, TypeRelOut, A02, 0).

len(TypeRelIn, TypeRelOut, A01, A02) :- iff(TypeRelIn, TypeRel1, A01, [_|A012]),
                                        iff(TypeRel1, TypeRel2, A012, L),
                                        iff(TypeRel2, TypeRel3, A02, s(A021)),
                                        iff(TypeRel3, TypeRel4, A021, S),
                                        len(TypeRel4, TypeRel5, A11, A12),
                                        iff(TypeRel5, TypeRel6, A11, L),
                                        iff(TypeRel6, TypeRelOut, A12, S).
```

Note that the variables are shared so that type relations may enter the clause at the head via `TypeRelIn`, cascade through the body, and exit the clause from the head via `TypeRelOut`. □

Evaluation of axiom goals is when and where the real work of obtaining type relations is done.

- If the axiom goal is `instance(TypeRelIn, TypeRelOut, Type, Const)`, then the type `Type` is associated with the constant type `Const`, and the association is added to the list of type relations passed in via `TypeRelIn` and then sent out via `TypeRelOut`.
- If the axiom goal is `iff(TypeRelIn, TypeRelOut, Type1, Type2)`, then the two type variables, `Type1` and `Type2`, are examined. If either of them are uninstantiated, then the two type variables are unified and so are `TypeRelOut` and `TypeRelIn` (i.e., no new type relations are added). If both `Type1` and `type2` are instantiated, then `Type1` and `Type2` are associated and the association is added to the list of type relations passed in via `TypeRelIn` and then sent out via `TypeRelOut`.

6 Results

The type inference program was implemented in Prolog using SICStus Prolog. The program was then ran on several toy programs. The programs and their outputs are shown in Figures 1 to 10. For program listings, numbers on the left indicate the clause numbers. For program outputs, numbers on the left merely indicate the line numbers. They are not part of the program nor part of the output.

Figure 1 shows the list length program we have been using as the example. As shown in Figure 2, the target program is loaded with `load_program/1`, and the type inference is started by calling `doit/0`. Information relevant to types is then retrieved by calling `typeinfo/0`. For `len`, our program was able to detect that the two arguments of `len` had recursive structures (lines 10 and 11 in Figure 2). For this small program, our program was successful.

The program was not so successful with the list reversal program in Figure 3. For this program, our program was able to detect the list structure in the first argument of `reverse/2` (using transitivity from lines 13 and 24 in Figure 4) and the first and second arguments of `reverse/3` (lines 23 and 24 in Figure 4), but not for the others. However, line 16 (and 17) in Figure 4 says that the second and the third arguments of `reverse/3` are of the same type. Thus, if we choose to extrapolate this, we could conclude that the list structure was also detected for the third arguments of `reverse/3`. Similar argument can be made for the second argument of `reverse/2` by line 1. Note, by the way, the large number of type equivalences (lines 6 to 9, 11 to 14, 16, 17, 19, and 20). Although these type relations are all crucial information inferred from the program, maybe our program should have tried some other means to keep all this information from being displayed.

This becomes a bigger problem for larger programs, as it can be witnessed in Appendix A. More would be said in Section 7.

Figure 5 is the list append program. This program is particularly tricky as the procedure, as it is written, works even if the second argument is not a list. Figure 6 shows that the program was able to detect the list structure for the first and third arguments (lines 13 and 14); as expected, not for the second argument. However, line 6 (and 7) says that the types of the second and third arguments are the same. Thus, again, if we choose to extrapolate this, we could conclude that the second argument also has a list structure.

Figure 7 is the list permuting program. This is also the first program that includes two procedures (apart from the list reversal program in Figure 3 where there were two closely related procedures). As shown in Figure 8, our program was able to detect list structures in both arguments of `permute` and in the third argument of `delete`. However, it was not able to detect the list structure in the second argument of `delete`; the most information it got out of the analysis was that the top functor was a list constructor (lines 10 and 18).

Figure 9 shows the quicksort program that was used to test our program. It is also the most complex program we used to test our program. Note the use of the successor function to represent non-negative integers. The number of lines printed by our program for type information was over 150 lines. Lines of interest are summarized in Figure 10; the whole output is shown in Appendix A. With the exception of `append`, our program was able to detect all the recursive structures.

7 Future Work

Our program could not infer all recursive constructs in term types. However, it was successful in most cases, at least with toy programs. An immediate problem with our program would be in its output processing; currently, there is barely any. Some filter is desired so that only the essential type information would be printed to the screen. Also, the use of type labels may need to be re-examined. The type labels lead to a large number of ground terms whose association need to be handled explicitly by the program. It would be better if some non-ground representation could be substituted so that type equivalence could be handled via unification.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[2] Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19, 20:199–260, 1994.

```
1 len([], 0).
2 len([X | Xs], s(S)) :- len(Xs, S).
```

Figure 1: List length program `len`.

```
1 | ?- load_program(len).

2 yes
3 | ?- doit.

4 yes
5 | ?- typeinfo.

6 typelabel$1.2 = 0.
7 typelabel$1.1 = [].
8 typelabel$2.2 = s(typelabel$1.2).
9 typelabel$2.1 = [_130|typelabel$1.1].
10 typelabel$2.2 = s(typelabel$2.2).
11 typelabel$2.1 = [_130|typelabel$2.1].
12 yes
```

Figure 2: Output for `len`.

```
1 reverse(Xs, RXs) :- reverse(Xs, [], RXs).
2 reverse([], Acc, Acc).
3 reverse([X | Xs], Acc, RXs) :- reverse(Xs, [X | Acc], RXs).
```

Figure 3: List reversal program `reverse`.

```
| ?- load_program(reverse).

yes
| ?- doit.

yes
| ?- typeinfo.
typelabel$1.2 = typelabel$2.3.
typelabel$2.3 = typelabel$1.2.
typelabel$1.1 = typelabel$2.1.
typelabel$2.1 = typelabel$1.1.
typelabel$2.2 = [].
typelabel$1.2 = typelabel$3.3.
typelabel$3.3 = typelabel$1.2.
typelabel$1.1 = typelabel$3.1.
typelabel$3.1 = typelabel$1.1.
typelabel$3.2 = [].
typelabel$2.3 = typelabel$2.2.
typelabel$2.2 = typelabel$2.3.
typelabel$2.1 = [].
typelabel$3.3 = typelabel$2.3.
typelabel$2.3 = typelabel$3.3.
typelabel$2.2 = [_130|typelabel$3.2].
typelabel$3.1 = [_130|typelabel$2.1].
typelabel$3.2 = [_130|typelabel$3.2].
typelabel$3.1 = [_130|typelabel$3.1].

yes
```

Figure 4: Output for `reverse`.

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

Figure 5: List append program `append`.

```
1  | ?- load_program(append).

2  yes
3  | ?- doit.

4  yes
5  | ?- typeinfo.
6  typelabel$1.3 = typelabel$1.2.
7  typelabel$1.2 = typelabel$1.3.
8  typelabel$1.1 = [].
9  typelabel$2.2 = typelabel$1.2.
10 typelabel$1.2 = typelabel$2.2.
11 typelabel$2.3 = [_130|typelabel$1.3].
12 typelabel$2.1 = [_130|typelabel$1.1].
13 typelabel$2.3 = [_130|typelabel$2.3].
14 typelabel$2.1 = [_130|typelabel$2.1].

15 yes
```

Figure 6: Output for `append`.

```
1  permute([], []).
2  permute([X | Xs], [Y | Ys]) :-
       delete(Y, [X | Xs], Zs),
       permute(Zs, Ys).

3  delete(X, [X | Xs], Xs).
4  delete(Y, [X | Xs], [X | Zs]) :-
       delete(Y, Xs, Zs).
```

Figure 7: List permute program `permute`.

```
1  | ?- load_program(permute).

2  yes
3  | ?- doit.

4  yes
5  | ?- typeinfo.
6  typelabel$1.2 = [].
7  typelabel$1.1 = [].
8  typelabel$3.3 = typelabel$1.1.
9  typelabel$1.1 = typelabel$3.3.
10 typelabel$3.2 = [_130|_131].
11 typelabel$2.2 = [typelabel$3.1|typelabel$1.2].
12 typelabel$2.1 = [_130|_131].
13 typelabel$3.3 = typelabel$2.1.
14 typelabel$2.1 = typelabel$3.3.
15 typelabel$2.2 = [typelabel$3.1|typelabel$2.2].
16 typelabel$4.3 = typelabel$1.1.
17 typelabel$1.1 = typelabel$4.3.
18 typelabel$4.2 = [_130|_131].
19 typelabel$2.2 = [typelabel$4.1|typelabel$1.2].
20 typelabel$4.3 = typelabel$2.1.
21 typelabel$2.1 = typelabel$4.3.
22 typelabel$2.2 = [typelabel$4.1|typelabel$2.2].
23 typelabel$4.1 = typelabel$3.1.
24 typelabel$3.1 = typelabel$4.1.
25 typelabel$4.3 = [_130|typelabel$3.3].
26 typelabel$4.3 = [_130|typelabel$4.3].

27 yes
```

Figure 8: Output for `permute`.

```
1  qsort([], []).
2  qsort([P | Xs], SXs) :-
       partition(Xs, P, Xs1, Xs2),
       qsort(Xs1, SXs1),
       qsort(Xs2, SXs2),
       append(SXs1, [P | SXs2], SXs).

3  partition([], _, [], []).
4  partition([X | Xs], P, [X | Ls], Bs) :-
       lessthan(X, P),
       partition(Xs, P, Ls, Bs).
5  partition([X | Xs], P, Ls, [X | Bs]) :-
       greaterequal(X, P),
       partition(Xs, P, Ls, Bs).

6  lessthan(0, s(_)).
7  lessthan(s(X), s(Y)) :- lessthan(X, Y).

8  greaterequal(0, 0).
9  greaterequal(s(_), 0).
10 greaterequal(s(X), s(Y)) :- greaterequal(X, Y).

11 append([], Ys, Ys).
12 append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

Figure 9: Quicksort program `qsort`.

```
    ...
18  typelabel$2.2 = typelabel$12.3.
    ...
45  typelabel$2.1 = typelabel$4.3.
    ...
58  typelabel$2.1 = typelabel$5.3.
    ...
72  typelabel$4.3 = [typelabel$6.1|typelabel$4.3].
73  typelabel$4.1 = [typelabel$6.1|typelabel$4.1].
    ...
83  typelabel$3.2 = typelabel$7.2.
    ...
87  typelabel$4.2 = typelabel$7.2.
88  typelabel$4.3 = [typelabel$7.1|typelabel$4.3].
89  typelabel$4.1 = [typelabel$7.1|typelabel$4.1].
    ...
91  typelabel$5.2 = typelabel$7.2.
    ...
102 typelabel$5.3 = typelabel$4.3.
103 typelabel$4.3 = typelabel$5.3.
    ...
110 typelabel$5.4 = [typelabel$8.1|typelabel$5.4].
111 typelabel$5.1 = [typelabel$8.1|typelabel$5.1].
    ...
122 typelabel$5.4 = [typelabel$9.1|typelabel$5.4].
123 typelabel$5.1 = [typelabel$9.1|typelabel$5.1].
    ...
125 typelabel$3.2 = typelabel$10.2.
    ...
129 typelabel$4.2 = typelabel$10.2.
    ...
133 typelabel$5.2 = typelabel$10.2.
    ...
140 typelabel$7.2 = s(typelabel$7.2).
141 typelabel$7.1 = s(typelabel$7.1).
    ...
150 typelabel$10.2 = s(typelabel$10.2).
151 typelabel$10.1 = s(typelabel$10.1).
    ...
159 typelabel$12.3 = [_130|typelabel$12.3].
160 typelabel$12.1 = [_130|typelabel$12.1].
    ...
```

Figure 10: Output for `qsort`.

A Complete Output for `qsort`

```
| ?- load_program(qsort4).

yes
| ?- doit.

yes
| ?- typeinfo.
typelabel$1.2 = [].
typelabel$1.1 = [].
typelabel$2.2 = typelabel$11.3.
typelabel$11.3 = typelabel$2.2.
typelabel$11.2 = [typelabel$3.2|typelabel$1.2].
typelabel$1.2 = typelabel$11.1.
typelabel$11.1 = typelabel$1.2.
typelabel$3.4 = typelabel$1.1.
typelabel$1.1 = typelabel$3.4.
typelabel$3.3 = typelabel$1.1.
typelabel$1.1 = typelabel$3.3.
typelabel$2.1 = [typelabel$3.2|typelabel$3.1].
typelabel$2.2 = typelabel$12.3.
typelabel$12.3 = typelabel$2.2.
typelabel$12.2 = [typelabel$3.2|typelabel$1.2].
typelabel$1.2 = typelabel$12.1.
typelabel$12.1 = typelabel$1.2.
typelabel$11.2 = [typelabel$3.2|typelabel$2.2].
typelabel$3.4 = typelabel$2.1.
typelabel$2.1 = typelabel$3.4.
typelabel$12.2 = [typelabel$3.2|typelabel$2.2].
typelabel$2.2 = typelabel$11.1.
typelabel$11.1 = typelabel$2.2.
typelabel$3.3 = typelabel$2.1.
typelabel$2.1 = typelabel$3.3.
typelabel$2.2 = typelabel$12.1.
typelabel$12.1 = typelabel$2.2.
typelabel$11.2 = [typelabel$4.2|typelabel$1.2].
typelabel$4.4 = typelabel$1.1.
typelabel$1.1 = typelabel$4.4.
typelabel$4.3 = typelabel$1.1.
typelabel$1.1 = typelabel$4.3.
typelabel$2.1 = [typelabel$4.2|typelabel$4.1].
typelabel$12.2 = [typelabel$4.2|typelabel$1.2].
```

```
40 typelabel$11.2 = [typelabel$4.2|typelabel$2.2].
41 typelabel$4.4 = typelabel$2.1.
42 typelabel$2.1 = typelabel$4.4.
43 typelabel$12.2 = [typelabel$4.2|typelabel$2.2].
44 typelabel$4.3 = typelabel$2.1.
45 typelabel$2.1 = typelabel$4.3.
46 typelabel$11.2 = [typelabel$5.2|typelabel$1.2].
47 typelabel$5.4 = typelabel$1.1.
48 typelabel$1.1 = typelabel$5.4.
49 typelabel$5.3 = typelabel$1.1.
50 typelabel$1.1 = typelabel$5.3.
51 typelabel$2.1 = [typelabel$5.2|typelabel$5.1].
52 typelabel$12.2 = [typelabel$5.2|typelabel$1.2].
53 typelabel$11.2 = [typelabel$5.2|typelabel$2.2].
54 typelabel$5.4 = typelabel$2.1.
55 typelabel$2.1 = typelabel$5.4.
56 typelabel$12.2 = [typelabel$5.2|typelabel$2.2].
57 typelabel$5.3 = typelabel$2.1.
58 typelabel$2.1 = typelabel$5.3.
59 typelabel$3.4 = [].
60 typelabel$3.3 = [].
61 typelabel$3.1 = [].
62 typelabel$4.4 = typelabel$3.4.
63 typelabel$3.4 = typelabel$4.4.
64 typelabel$4.2 = typelabel$3.2.
65 typelabel$3.2 = typelabel$4.2.
66 typelabel$6.2 = typelabel$3.2.
67 typelabel$3.2 = typelabel$6.2.
68 typelabel$4.3 = [typelabel$6.1|typelabel$3.3].
69 typelabel$4.1 = [typelabel$6.1|typelabel$3.1].
70 typelabel$6.2 = typelabel$4.2.
71 typelabel$4.2 = typelabel$6.2.
72 typelabel$4.3 = [typelabel$6.1|typelabel$4.3].
73 typelabel$4.1 = [typelabel$6.1|typelabel$4.1].
74 typelabel$4.4 = typelabel$5.4.
75 typelabel$5.4 = typelabel$4.4.
76 typelabel$4.2 = typelabel$5.2.
77 typelabel$5.2 = typelabel$4.2.
78 typelabel$6.2 = typelabel$5.2.
79 typelabel$5.2 = typelabel$6.2.
80 typelabel$4.3 = [typelabel$6.1|typelabel$5.3].
81 typelabel$4.1 = [typelabel$6.1|typelabel$5.1].
82 typelabel$7.2 = typelabel$3.2.
```

```
83  typelabel$3.2 = typelabel$7.2.
84  typelabel$4.3 = [typelabel$7.1|typelabel$3.3].
85  typelabel$4.1 = [typelabel$7.1|typelabel$3.1].
86  typelabel$7.2 = typelabel$4.2.
87  typelabel$4.2 = typelabel$7.2.
88  typelabel$4.3 = [typelabel$7.1|typelabel$4.3].
89  typelabel$4.1 = [typelabel$7.1|typelabel$4.1].
90  typelabel$7.2 = typelabel$5.2.
91  typelabel$5.2 = typelabel$7.2.
92  typelabel$4.3 = [typelabel$7.1|typelabel$5.3].
93  typelabel$4.1 = [typelabel$7.1|typelabel$5.1].
94  typelabel$5.3 = typelabel$3.3.
95  typelabel$3.3 = typelabel$5.3.
96  typelabel$5.2 = typelabel$3.2.
97  typelabel$3.2 = typelabel$5.2.
98  typelabel$8.2 = typelabel$3.2.
99  typelabel$3.2 = typelabel$8.2.
100 typelabel$5.4 = [typelabel$8.1|typelabel$3.4].
101 typelabel$5.1 = [typelabel$8.1|typelabel$3.1].
102 typelabel$5.3 = typelabel$4.3.
103 typelabel$4.3 = typelabel$5.3.
104 typelabel$8.2 = typelabel$4.2.
105 typelabel$4.2 = typelabel$8.2.
106 typelabel$5.4 = [typelabel$8.1|typelabel$4.4].
107 typelabel$5.1 = [typelabel$8.1|typelabel$4.1].
108 typelabel$8.2 = typelabel$5.2.
109 typelabel$5.2 = typelabel$8.2.
110 typelabel$5.4 = [typelabel$8.1|typelabel$5.4].
111 typelabel$5.1 = [typelabel$8.1|typelabel$5.1].
112 typelabel$9.2 = typelabel$3.2.
113 typelabel$3.2 = typelabel$9.2.
114 typelabel$5.4 = [typelabel$9.1|typelabel$3.4].
115 typelabel$5.1 = [typelabel$9.1|typelabel$3.1].
116 typelabel$9.2 = typelabel$4.2.
117 typelabel$4.2 = typelabel$9.2.
118 typelabel$5.4 = [typelabel$9.1|typelabel$4.4].
119 typelabel$5.1 = [typelabel$9.1|typelabel$4.1].
120 typelabel$9.2 = typelabel$5.2.
121 typelabel$5.2 = typelabel$9.2.
122 typelabel$5.4 = [typelabel$9.1|typelabel$5.4].
123 typelabel$5.1 = [typelabel$9.1|typelabel$5.1].
124 typelabel$10.2 = typelabel$3.2.
125 typelabel$3.2 = typelabel$10.2.
```

```
126 typelabel$5.4 = [typelabel$10.1|typelabel$3.4].
127 typelabel$5.1 = [typelabel$10.1|typelabel$3.1].
128 typelabel$10.2 = typelabel$4.2.
129 typelabel$4.2 = typelabel$10.2.
130 typelabel$5.4 = [typelabel$10.1|typelabel$4.4].
131 typelabel$5.1 = [typelabel$10.1|typelabel$4.1].
132 typelabel$10.2 = typelabel$5.2.
133 typelabel$5.2 = typelabel$10.2.
134 typelabel$5.4 = [typelabel$10.1|typelabel$5.4].
135 typelabel$5.1 = [typelabel$10.1|typelabel$5.1].
136 typelabel$6.2 = s(_131).
137 typelabel$6.1 = 0.
138 typelabel$7.2 = s(typelabel$6.2).
139 typelabel$7.1 = s(typelabel$6.1).
140 typelabel$7.2 = s(typelabel$7.2).
141 typelabel$7.1 = s(typelabel$7.1).
142 typelabel$8.2 = 0.
143 typelabel$8.1 = 0.
144 typelabel$9.2 = 0.
145 typelabel$9.1 = s(_131).
146 typelabel$10.2 = s(typelabel$8.2).
147 typelabel$10.1 = s(typelabel$8.1).
148 typelabel$10.2 = s(typelabel$9.2).
149 typelabel$10.1 = s(typelabel$9.1).
150 typelabel$10.2 = s(typelabel$10.2).
151 typelabel$10.1 = s(typelabel$10.1).
152 typelabel$11.3 = typelabel$11.2.
153 typelabel$11.2 = typelabel$11.3.
154 typelabel$11.1 = [].
155 typelabel$12.2 = typelabel$11.2.
156 typelabel$11.2 = typelabel$12.2.
157 typelabel$12.3 = [_130|typelabel$11.3].
158 typelabel$12.1 = [_130|typelabel$11.1].
159 typelabel$12.3 = [_130|typelabel$12.3].
160 typelabel$12.1 = [_130|typelabel$12.1].

161 yes
```