LIFETIME ANALYSIS FOR PROCEDURE PARAMETERS

U. Kastens, M. Schmidt

Universität-GH Paderborn

ABSTRACT

Procedural abstraction is an important concept for functional and imperative programming as well. Compilers should implement procedures and calls as efficient as possible. This paper addresses techniques for both memory and runtime improvements of procedure calls. Usual implementations allocate storage for local objects of procedures (i.e. parameters, result and local variables) either in activation records on a runtime stack in the case of recursive procedures or in a global storage area otherwise. We show that objects of recursive procedures can also be allocated globally, if certain conditions on their lifetime are met. We present a method for systematic lifetime analysis based on a context-free grammar which describes the calling structure of the program. It is introduced for a simple functional language and then extended to cover further common language properties. Furthermore it is shown that the method can nicely be combined with context dependent improvements of the compilation of calling sequences, e.g. for tail recursive calls.

INTRODUCTION

The usual storage allocation technique for hierarchical structured languages is a runtime stack for activation records each containing the local objects of a procedure or block instance. If a compiler can prove that a certain procedure is not called recursively its parameters and its local variables can be statically allocated in a global area. No activation record is needed on the stack in that case.

In this paper we present a method for lifetime analysis of parameters and local variables of procedures. The result is that certain objects of recursively called procedures can be allocated as global variables, too. This more elaborated approach yields better results than the rough distinction between recursive and non-recursive procedures.

Furthermore this technique is applicable to objects for procedure call implementation, like return address or links to static environments usually contained in activation records. Conventional compiling techniques [WG 84, AU 78] use fixed calling sequences and activation record structures. We propose here to analyze procedure properties in order to reduce the data space on the runtime stack and to improve the calling sequences individually for procedures and their local objects.

The techniques presented here are applicable to compilation of functional languages as well as imperative ones. Of course one can expect more significant results if programs contain many recursive procedures as in functional languages. Most of the restrictions we impose on language constructs for the application of our technique are not essential (except the exclusion of aliasing), but are introduced for ease of presentation and dropped in later sections.

Our technique of lifetime analysis for procedure objects is based on context-free grammars. Its sentences describe the definition and use of parameter instances in any possible program execution. This idea was inspired by comparable techniques developed for storage reduction in attribute evaluators which are automatically generated by the GAG-System [KHZ 82, Ka 84].

This paper is organized as follows: Section 2 introduces the basic method of lifetime analysis. The area of its application is extended to functional parameters in section 3 and to nested procedures in section 4. Global implementation of return addresses and improvements of calling sequences are treated in section 5.

LIFETIME ANALYSIS OF PROCEDURE PARAMETERS

We present a method for analysis of the lifetime of parameters and local variables of procedures. The goal is to state a condition which decides for a single local object whether all its instances in recursive calls can be implemented by one global variable instead of a stack element for each instance.

Our method is based on conditions over the language of a context-free grammar. The procedure bodies are transformed into a set of productions where calls are mapped to the nonterminals and definition and use of objects become the terminal symbols. A derivation for such a grammar corresponds to one of the possible dynamic calling structures of the program. Its sentences describe start and end of the parameters' lifetime. A condition over the language decides whether the lifetimes of all instances of certain objects are disjoint in any possible calling structure. These objects can be allocated statically as global variables.

The method is restricted to objects which are not subject to aliasing. For ease of presentation we assume for the moment that functional parameters do not occur, procedures are not nested and there is no access to non-local objects. These concepts are treated in sections 3 and 4. The notation introduced below does not include loops for imperative languages. We assume that they are removed by equivalent recursive procedures.

For each procedure f three sets of objects are considered: input parameters $I(f)$, output parameters $O(f)$, and local objects $V(f)$. We assume a symmetrical parameter passing mechanism for input and output parameters. It differs slightly from usual techniques. But the techniques call by value, by reference, by result and function result passing can be mapped to this more general technique: Input parameters are defined (not necessarily immediately) before the call and are used in the called procedure. Output parameters are defined in the called procedure and used in the caller after the call. There is no special case for function results; they are treated as output parameters.

A procedure body is a sequence of basic operations structured by nested or sequenced conditionals. Hence, we assume that in the case of applicative programs, an evaluation order is already determined. For ease of further analysis we start from an abstract procedure notation describing only definition (D), use (U) and procedure call ($proc_id$) as stated in the following syntax:

```

proc          ::= proc identifier (local_objects) : op_seq.
local_objects ::= identifier {, identifier}
op_seq       ::= operation*.
operation    ::= if op_seq then op_seq else op_seq fi | proc_id | basic_op.
basic_op     ::= D | U.

```

As we do not yet consider access to global objects no notation for nested procedures is provided. The D and U symbols are indexed with the name of the object (e.g. D_a for a definition of a). We assume that throughout the program all objects are uniquely named. If several calls of the same procedure occur in one context or if a procedure is called directly recursive then references to their objects are distinguished by number indices, e.g.:

$$\dots D_a^1 \dots f_1 \dots D_a^2 \dots f_2 \dots$$

An assignment like

$$a := b + c$$

is written in our notation

$$U_b U_c D_a$$

or

$$U_c U_b D_a$$

expressing that b and c are used to define a .

A call of a function f with input parameters a and b and a single result like

$$c := f(x, y + z)$$

appears as a procedure with an additional output parameter, say r , assigned to c :

$$U_x D_a U_y U_z D_b f U_r D_c$$

where U_r stands for the use of the result of the call. If an object like a local variable is redefined in a procedure, D represents the position of its first definition.

A set of procedures given in the above notation is transformed into a lifetime grammar $G_L = (N_L, T_L, P_L, S_L)$ by the three following steps. The set of nonterminals N_L is the set of procedure names. The terminals are $T_L = \{D_a, U_a, L_a \mid a \text{ is an object name}\}$. These symbols may be indexed as described above. The start symbol S_L is the name of that procedure which is called as the root of the program. The productions are a straight forward transcription of the procedure definitions with the procedure name as lefthand side. In order to express all possible execution paths through a procedure body, each conditional clause

if u then v else w fi

is transformed into the extended BNF string

$u(v \mid w)$

If we project on our example to the set $\{y, r\}$ we will get the following grammar, which obviously meets the globalization condition:

```
(main → D ack L)
ack →   L D   |
        L D ack L D   |
        L D ack L D ack L D
```

The remarkable result is, that the second argument and the result of the Ackermannfunction are globalizable in a common variable.

FUNCTIONAL PARAMETERS

In the previous section we excluded functional parameters for ease of presentation. In fact our method of lifetime analysis can be expanded to the treatment of functional parameters in a straight forward manner. Of course the simple question whether the value of such a parameter (the function reference) can be stored in a global variable is answered by the lifetime analysis in the same way as for parameters of other types. The following extensions address the problem that lifetimes of the objects of procedures used as actual parameters are determined by the context of the calls of the corresponding formal procedures. Even recursion may be

hidden by the use of functional parameters.

In our lifetime analysis the data dependency for the choice of a certain alternative is not considered. Consequently we assume that a formal functional parameter may be replaced by any correspondingly occurring actual parameter. That will be expressed in our lifetime grammar by adding certain chain productions. For ease of description we assume that functional values used in parameter assignments are denoted by declared names. (Remind that we do not yet consider nested procedures.)

In order to avoid peculiar notations let the names of all procedures and formal functional parameters be different. Let p be a procedure with a functional parameter f , and q the name of an arbitrary declared or formal procedure. If f is an input parameter a call of p may have the form $p(q)$ and f may be called (or further passed as parameter) in the body of p . If f is an output parameter the body of p contains parameter assignments of the form $f:=q$ and f may be called (or further passed as parameter) after a call of p . For the construction of our lifetime grammar G_L calls of f are treated like calls of declared procedures: They are transformed into occurrences of the nonterminal f . For each parameter assignment to f (input as well as output) a chain production $f::=q$ is added to P_L . By this means the language $L(G_L)$ reflects all possible sequences of calls in any program execution as required for lifetime analysis. (Possibly occurring cyclic chain productions do not effect the decision of the globalization condition on the base of $L(G_B)$).

ACCESS TO GLOBAL OBJECTS

For our basic method we assumed a flat program structure, i.e. a program consists of a set of procedures which are not nested, and there are no explicit global variables referred to. In the following we consider a systematic transformation which yields that structure for any program with nested procedures in order to apply our lifetime analysis technique. The results are twofold: On the one hand one can follow that transformation and hope that the effect of globalization will pay off for the introduced overhead. On the other hand one can decide to implement objects which are accessed from inner procedures in activation records. In that case the optimization results can be used to eliminate static links from activation records.

Assume an arbitrary nested program like (4.1). In that example a, b, c are input parameters and x, y, z are output parameters. In order to keep the example simple we omitted any operation on the values except parameter passing. In the body of r there is a non-local access to b .

```

(4.1) proc p (a, x) :
  begin
    proc q (b, y) :
      begin
        proc r (c, z) :
          begin
            if b
            then q(c, z)
            else z := c
            fi
          end;
          r(b, y)
        end;
        q(a, x)
      end
    end
  end

```

```

(4.2) proc p(a, x) :
  begin
    proc q(b, y) :
      proc r(c, z; e) :
        begin
          if e
          then q(c, z)
          else z := c
          fi
        end;
        r(b, y; b)
      end;
      q(a, x)
    end
  end

```

```

(4.3) proc p(a, x)      : q (a, x);
      proc q(b, y)      : r(b, y; b);
      proc r(c, z; e)   : if e then q(c, z) else z:=c fi

```

In order to achieve a flat program structure we replace each non-local object used in a procedure body by an additional input parameter. (In our example in the procedure `r` `b` is replaced by the new parameter `e`.) In the calls of that procedure the substituted object is added as the corresponding actual parameter.

These transformations are repeated until no non-local accesses remain. (4.2 is the result for our example.) Now the program structure can be flattened without changing the semantics (4.3).

If we apply lifetime analysis to the transformed program the results for the formerly globally accessed objects and the introduced parameters depend on the points of the non-local accesses in relation to the procedure calls. The productions of G_L for 4.3 are (disregarding a and x)

$$\begin{aligned}
 (4.4) \quad p &::= D_b q L_y \\
 q &::= D_c L_b D_e r L_x D_y \\
 r &::= L_e L_c D_b q L_y D_x \\
 r &::= L_e L_c D_x
 \end{aligned}$$

One easily verifies that $L(G_B) \subseteq (DL)^*$ for $B=\{b\}$, $B=\{e\}$, and $B=\{b,e\}$. The latter case means that the object b which was referred to non-locally can be mapped to a global variable together with the parameter substituted for it. The identical parameter assignment of b to e in the call of r can be eliminated.

The result of our example (b and e can be mapped to a single global variable) is the best of several possible cases. Compared with conventional implementation techniques the following advantages are gained. No space for b is needed in the activation record of q ; in the activation record of r no static link to the environment is needed (provided there is no other non-local access in r); and that static link need not be computed on entry of r . In this sense the comparison will show less savings in the cases where $L(G_B) \subseteq (DL)^*$ does not hold for $B=\{b,e\}$ but for only $B=\{b\}$, only $B=\{e\}$, $B=\{b\}$ and $B=\{e\}$. Only if neither of these conditions hold the conventional implementation is preferable to the transformed solution with b and e on the activation record stack.

This reasoning leads to a second interpretation of the above technique: Assume that procedures like q have an activation record allocated on the stack, and that objects in it are accessed from embedded procedures like r . Then in conventional implementation r needs a link to its static environment leading to an activation record of q . That static predecessor link is allocated in r 's activation record. It is assigned at the call of r . So we can interpret the parameter e introduced in the transformation (4.2) as follows: It represents the static link in the activation record of r used for non-local access of q 's objects. Then the result of lifetime analysis decides whether the static link can be implemented by a global variable (if $L(G_B) \subseteq (DL)^*$ for $B=\{e\}$).

IMPROVEMENTS OF CALLING SEQUENCES

In this section we shall use results of the lifetime analysis in order to improve calling sequences for procedures. In particular we consider the return address for procedure calls. It can be treated as an input parameter of the called procedure, which is defined at the calling point, stored in the activation record on the runtime stack, and used for the return jump. Lifetime analysis applied to this structure obviously shows that the return address can be stored in a global variable if and only if the procedure is not recursive. But even for recursive procedures one can get better results in certain cases: If a procedure body contains a call at its end, and if the output parameters (or function results) are already mapped to global variables, the calling sequence can be transformed such that the return leads directly to a previous caller avoiding to wind up the intermediate levels. (In [Ha 75] it is termed "goto type expression".) This transformation yields shorter and faster calling sequences and saves space on the runtime stack. In the case of directly tail-recursive procedures it even removes recursion. This effect is comparable to certain program transformation techniques as described e.g. in [DB 76, BW 82]. Furthermore after that transformation lifetime analysis may show that the return address can be implemented globally even for certain recursive procedures.

We consider only three distinct objects in an activation record: For ease of representation we will treat the output parameters and the input parameters as one object each. The activation record for each procedure contains the objects in the order output parameters, return address, input parameters. In contrast to conventional implementations of activation records [WG 84] we do not consider links to static and dynamic predecessors. For static predecessors refer to section 4. The dynamic link can either be treated as an implicit input parameter or it can be eliminated at all if local objects are addressed relative to the top of the stack.

For the following discussion of calling sequences we use an abstract notation which can easily be mapped to instructions of a stack machine. We will use a generalized assembler language for a stack assembler with the following operations:

allocate	reserve space for an object on top of the stack
assign	set value of an object, its space on the stack is previously allocated
use	get value of an object for the subsequent push or assign
push	push object onto the stack

pop free object on top of the stack
 jump go to address for further execution

This operations correspond to certain sequences of D and U namely assign $\equiv$ D, use $\equiv$ U and push $\equiv$ D of some objects.

We refer to the output and input parameters and the return address of a procedure f as out_f , in_f and ra_f respectively. Then the calling sequence for a procedure q in a procedure p is as follows:

```

    allocate  $out_q$    ; output parameters
    push A      ; return address
    use  $in_p$       ; compute value
    push  $in_q$     ; store input parameters
    jump q
A:   use  $out_q$     ; use the results of q

```

This corresponds in the abstract procedure notation to

```

            $D_{ra_q}$        $U_{in_p}$        $D_{in_q}$       q       $U_{out_q}$ 
push A   use  $in_p$    push  $in_q$       use  $out_q$ 

```

The return sequence of a procedure p will look like

```

p:  .
    .
    .
    assign  $out_p$ 
    pop  $in_p$ 
    pop  $ra_p$ 
    jump  $ra_p$ 

```

This in turn corresponds to

```

            $D_{out_p}$        $U_{ra_p}$ 

```

assign out_p jump ra_p

On entry of procedure q called by p the stack containing the activation records looks as follows:

Using this generalized notation we can give a calling sequence for the case that the call of q is the last operation in p :

```

p:  .
    .
    .
    allocate  $out_q$ 
    push A       $D_{ra_q}$ 
    use  $in_p$      $L_{in_p}$ 
    push  $in_q$     $D_{in_q}$ 
    jump q
    use  $out_q$      $L_{out_q}$ 
    assign  $out_p$   $D_{out_p}$ 
    pop  $out_q$ 
    pop  $in_p$ 
    pop  $ra_p$ 
    jump  $ra_p$     $L_{ra_p}$ 

```

Where q ends as expected in

```

q:  .
    .
    .
    assign  $out_q$   $D_{out_q}$ 

```

```

pop  $in_q$ 
pop  $ra_q$ 
jump  $ra_q$        $L_{ra_q}$ 

```

Now we introduce the two facts that out_p and out_q are globalizable in a single variable and that the assignment operation (expressed in use_{out_q} ; $assign_{out_p}$) is the identity.

The calling sequence in p shrinks to:

```

p:      . .
        .
        .
        push A       $D_{ra_q}$ 
        use  $in_p$      $L_{in_p}$ 
        push  $in_q$     $D_{in_q}$ 
        jump q
A:      pop  $in_p$ 
        pop  $ra_p$ 
        jump  $ra_p$      $L_{ra_p}$ 

```

Looking at this sequence we recognize that the last use of in_p is (in the worst case) in the definition of in_q . So we can pop in_p just before we push in_q . Now we investigate the return addresses. The only remaining action after the call of q is to pop ra_p and return. This action can be removed by giving the procedure q the return address of p .

This results in the following calling sequence for q in p .

```

p:      .
        .
        .
        pop  $in_p$ 
        use  $in_p$ 
        push  $in_q$ 
        jump q

```

The saving in the code is obvious. The saving in the stackdepth is easily shown by a comparison of the contents of the stack on entry of procedure q before the

optimization and afterwards:

These improvements allow a further investigation of the return address. This could result in a globalization of some return addresses even of recursive functions, for example tail recursive functions. The improvements also cover the transformation of tail recursive functions into loops. In this case it is possible to replace the pop-push-sequence at the end of the calling procedure by an assignment in place. More generally spoken all goto type function calls of [Ha 75], [St 77] are transformed into branches. These branches are all either to an explicitly mentioned address or to an address on top of the stack. So no complicated and expensive returns from procedures are necessary in the optimized cases.

EXAMPLE

To illustrate the algorithms of the previous sections we will apply them to the well known Ackermann-Function.

The Ackermann-Function is defined by

$$ack(x, y) = \begin{cases} y+1, & \text{if } x=0 \\ ack(x-1, 1), & \text{if } x \neq 0 \text{ and } y=0 \\ ack(x-1, ack(x, y-1)), & \text{if } x \neq 0 \text{ and } y \neq 0 \end{cases}$$

This yields directly the following Pascal-Function:

```
function ack (x, y : integer) : integer;
begin
```

```

if x=0
then ack := y+1
else if y=0
    then ack := ack(x-1, 1)
    else ack := ack(x-1, ack(x, y-1))
end;
procedure main; begin write (ack(a,b)) end

```

The next step is to write the calling sequences and to convert the operations in the lifetime grammar G_L . Then for every object a all but the last occurrences of U_a are deleted, the last is renamed L_a . This results in a grammar with the following productions:

```

main   →   $D_x D_y$  ack  $L_r$ 
ack    →   $L_x L_y D_r$ 
         $L_y L_x D_x D_y$  ack  $L_r D_r$ 
         $D_x L_x D_x L_y D_y$  ack  $L_r D_y$  ack  $L_r D_r$ 

```

These productions now are projected to the sets $\{y\}$, $\{r\}$ and $\{y, r\}$. In all three cases the projections yield a grammar, which produces a subset of $(DL)^*$. So the result and the second parameter of the Ackermann-Function can be mapped to a single global variable. In Pascal that looks like

```

var global : integer
procedure ack (x);
begin
    if x=0
    then global := global + 1
    else if global = 0
        then begin
            global := 1
            ack(x-1) {id}
        end else begin
            global := global - 1;
            ack(x); {id}
            ack(x-1) {id}
        end
    end;
procedure main;
begin global := b; ack(a); write(global) end

```

At the positions marked by {id} identical assignments ($\text{global} := \text{global}$, for y and the result r) are omitted.

This Pascal-Procedure contains still two "goto type" function calls. These can be eliminated by the method presented in section 5, because the output parameters are globalizable and the call of *ack* is the last action. So our algorithms eliminate two of the three function calls and two objects of each activation record on the stack. The resulting procedure is shown below. It is written in Pascal only for ease of presentation. An equivalent machine language program could be produced by a compiler applying our techniques.

```

var global : integer;
procedure ack (x : integer);
label 1;
begin
  1:
    if x=0
    then global := global + 1
    else if global = 0
    then begin
      global := 1
      x:=x-1,
      goto 1
    end else begin
      global := global - 1,
      ack(x),
      x:=x-1;
      goto 1
    end;
end;

procedure main;
begin global := b; ack(a); write(global) end

```

The following table compares the runtimes and maximum recursion depth of these two versions of the Ackermannfunction. The storage reduction on the stack results in the reduction of the recursive depth and in the reduction of the activation record by two integer objects.

	normal version	optimized version
	runtime	runtime
	(sec)	(sec)
ack(3,3)	0.11	0.11
ack(3,4)	0.50	0.35
ack(3,5)	1.95	1.45
ack(3,6)	7.90	5.75
ack(3,7)	31.96	23.03
ack(3,8)	2:09.40	1:32.41

REFERENCES

- [AU 78] Aho, A.V., Ullman, J. D.: Principles of Compiler Design. Addison Wessley (1978)
- [BW 82] Bauer, F.L., Wössner, H.: Algorithmic Language and Program Development. Springer Verlag (1982)
- [DB 76] Darlington, J., Burstall, R.M.: A System Which Automatically Improves Programs. Acta Informatica 6 (1976), pp. 41-60
- [Ha 75] Haskell, R.: Efficient Implementation of a Class or Recursively Defined Functions. The Computer Journal, Vol. 18 No. 1, pp. 23-29
- [Ka 84] Kastens, U.: The GAG-System - A Tool for Compiler Construction. in Lorho, B. (ed): Methods and Tools for Compiler Construction, Cambridge University Press, 165-181
- [KHZ 82] Kastens, U., Hutt, B., Zimmermann, E.: GAG: A Practical Compiler Generator. LNCS 141 (1982)
- [St 77] Steele, G.L.: Debunking the Expensive Procedure Call" Myth..... Proc. of the ACM Nat. Conf., pp. 153-162 (1977)
- [WG 84] Waite, W.M., Goos, G.: Compiler Construction. Springer Verlag (1984)