TAPENADE for C

Valérie Pascual and Laurent Hascoët

INRIA, TROPICS team, 2004 route des Lucioles 06902 Sophia-Antipolis, France
{Valerie.Pascual,Laurent.Hascoet}@sophia.inria.fr

Summary. We present the first version of the tool TAPENADE that can differentiate C programs. The architecture of TAPENADE was designed from the start to be language independent. We describe how this choice made adaption to C easier. In principle, it needed only a new front-end and back-end for C. However we encountered several problems, in particular related to declarations style, include files, parameter-passing mechanism, and extensive use of pointers. We describe how we addressed these problems, and how the resulting improvements also benefits to differentiation of Fortran programs.

Key words: Automatic Differentiation, Software Tools, Program Transformation, TAPENADE, C

1 Introduction

We present the first version of the tool TAPENADE [**?**] that can differentiate C programs [**?**, **?**]. TAPENADE is an Automatic Differentiation (AD) tool [**?**] that produces differentiated programs by source analysis and transformation. Given the source of a program, along with a description of which derivatives are needed, TAPENADE creates a new source that computes the derivatives. TAPENADE implements tangent differentiation and reverse differentiation.

Right from the start in 1999, TAPENADE was designed as mostly independent from the source language, provided it is imperative. Figure 1 summarizes this architecture. The differentiation engine is built above a kernel that holds an abstract internal representation of programs and that runs static analysis (e.g. data-flow). This composes TAPENADE strictly speaking. The internal representation does not depend on the particular source language. This architecture allows the central module to forget about mostly syntactic details of the analyzed language, and to concentrate on the semantic constructs. Programs are represented as Call Graphs, Control Flow Graphs [**?**], and Basic Blocks linked to Symbol Tables. Syntax Trees occur only at the deepest level: elementary statements.

In addition to the TAPENADE kernel, for any source language there must be separate front- and back-end. They exchange programs with TAPENADE's kernel via an abstract Imperative Language called IL. Initially, there was only a front- and a back-end for Fortran77. These were followed by a front- and a back-end for Fortran95 [**?**]. Now is the time for C.

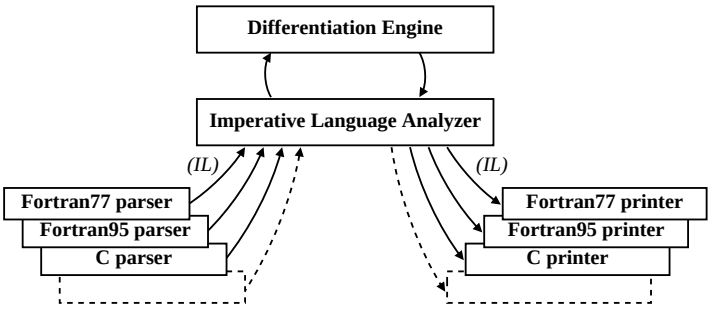


Fig. 1. Architecture sketch of TAPENADE

There are other AD tools that differentiate C programs e.g. ADOL-C [**?**], ADIC [**?**], or TAC++ [**?**]. Some of these tools, e.g. ADOL-C, have no Fortran equivalent as they rely on operator overloading. For the others, the idea of sharing a part of their implementation with a Fortran equivalent came gradually. As a C equivalent of ADIFOR, ADIC did mention this possibility. The architecture of TAPENADE was explicitly designed for this objective since its start in 1999. The XAIF concept [**?**] at the basis of the OpenAD [**?**] environment also aims at promoting this sharing. OpenAD contains a new version of ADIC as its component for C. To our knowledge, TAC++ shares algorithms but no implementation with TAF.

The architecture of TAPENADE should make extension for C relatively easy. Ideally one should only write a new front- and back-end. In reality things turned out to be slightly more complex. Still, our revision control tool tells us that, on 120 000 lines of JAVA code of TAPENADE, less than 10% of the total have been modified for C handling. We consider this a very positive sign of the validity of the architecture. In particular C structured types came for free, as they were already handled by the type representations needed by Fortran95. The same holds for the C variable scoping mechanism, and for most control structures. Pointers are already present in Fortran95, but only with C are pointers used at such a large scale so we feel this is the right context to present our alias analysis.

This paper discusses the features of C that required choices, improvements, and new developments in TAPENADE. We emphasize that many of the improvements actually concern erroneous design choices in TAPENADE, that often resulted from having implicitly Fortran in mind when making these choices. We believe the new choices are more general, make the tool more abstract and safe, and benefit even to the differentiation of Fortran. In the sequel, we will refer to the TAPENADE just before considering C as "the old TAPENADE", whereas the current version resulting from this is called "the new TAPENADE". The changes that we describe fall into the following categories: Section 2 briefly describes the external front- and back-end for C. Section 3 deals with the new handling of declaration statements, yielding a major change regarding include files. Section 4 discusses the parameter-passing mechanism. Section 5 describes the flow-sensitive alias analysis for a precise pointer destinations information. In section 6, we summarize the current status of TAPENADE for C, discuss some remaining limitations, and evaluate the cost of the more distant extension to object-oriented languages.

2 Front-end and Back-end for C

The new front-end for C is composed of three successive parts:

1. a preprocessor (actually the standard CPP preprocessor for C) inlines `#include` directives, and processes macro definitions `#define` and conditional inclusions `#if`, `#ifdef`... However we keep placeholders for the beginning and end of include files. These placeholders are kept through the complete differentiation process, allowing TAPENADE to generate shorter code that explicitly makes `#include` calls. On the other hand the other directives e.g. `#define`, `#ifdef`, are not reinstalled in the differentiated program.
2. a parser performs the true lexical and syntactic analysis of the preprocessed source. It is based on the `antlr` parser generator [**?**]. It supports the Standard C language [**?**, **?**]. It returns an abstract syntax tree.
3. a translator turns the syntax tree into a serialized IL tree, ready to be transferred into TAPENADE using the same protocol as the other front-ends.

The back-end for C translates IL trees into C source code, using pretty much the same algorithm as the Fortran back-ends. For spacing and indenting, it implements the recommended style for C [**?**]. In contrast, it does not alter the naming conventions (e.g. capitalization) of the original program. The back-end uses the include placeholders to reinstall `#include` directives whenever possible. This mechanism also benefits to the Fortran back-ends.

Unlike the Fortran front- and back-ends, those for C are compiled into JAVA code exclusively, thus making the distribution process easier on most platforms.

3 Declaration Statements

In the old TAPENADE, the internal representation held the type information only as entries in the symbol tables. The original declaration statements were digested and thrown away by the analysis process. Therefore on the way back, the produced programs could only create declarations in some standard way, unrelated to the order and style of the original source. Consequently, differentiated declarations were harder to read, comments were lost or at best floated to the end of the declaration section, and include calls were systematically expanded.

This model is not acceptable for C. Include files are commonplace, and they are long and complex. Inlining `stdio.h` is not an option! Also, declarations may contain initializations, which need to be differentiated as any other assignment statement. Like for e.g. JAVA source, declarations can be interleaved with plain statements and the order does matter.

In the new TAPENADE, the syntax trees of declaration statements are kept in the Flow Graph as for any other statement. They are used to build the symbol tables but are not thrown away. During differentiation, declaration statements are differentiated like others. The order of declarations in the differentiated source matches that of the original source. The same holds for the order of modifiers inside a declaration, like in `int const i`.

Relative ordering of differentiated statements is worth mentioning. In tangent differentiation mode, the differentiation of plain assignments is systematically placed *before* the original assignment. This is because the assignment may overwrite a variable used in the right-hand side. This never happens for declarations, though, because assignments in declarations are only initializations. This ordering constraint is relaxed. On the other hand, one declaration can gather several successive initializations that may depend on one another. The differentiated initialization may depend on one of the original initializations, and in this case the differentiated declaration statement must be placed *after*. In the reverse mode of AD, differentiation of a declaration with initialization cannot result in a single statement: the differentiated declaration must go to the top of the procedure, and the differentiated initialization must go to the end of

the procedure. There is no fixed rule for ordering and we resort to the general strategy already present in TAPENADE namely, build a dependency graph between differentiated statements, including declarations and initializations.

Given for instance the following procedure:

```
void test(float x, float y, float *z)
{
    /* comment on declaration */
    float u = x * 2, v = y * u;
    u = u * v;
    float w = *z * u;
    /* comment on statement */
    *z = w * (*z);
}
```

the new TAPENADE produces the following tangent differentiated procedure:

```
void test_d(float x, float xd, float y,
            float yd, float *z, float *zd)
{
    /* comment on declaration */
    float u = x*2, v = y*u;
    float ud = 2*xd, vd = yd*u + y*ud;
    ud = ud*v + u*vd;
    u = u*v;
    float w = *z*u;
    float wd = *zd*u + *z*ud;
    /* comment on statement */
    *zd = wd*(*z) + w*(*zd);
    *z = w*(*z);
}
```

whereas the reverse differentiated procedure has split differentiation of declarations with initialization:

```
void test_b(float x, float xb, float y,
            float yb, float *z, float *zb)
{
    /* comment on declaration */
    float u = x*2, v = y*u;
    float ub, vb;
    /* ... code stripped out for clarity ... */
    ub = y*vb + v*ub;
    yb = yb + u*vb;
    xb = xb + 2*ub;
}
```

Preserving declarations order allows TAPENADE to reinstall most `#include` directives in the generated code. The example in Fig. 3 illustrates this for C. We already mentioned that the preprocessor keeps track of the include files, so that TAPENADE can label declarations with their origin and propagate these labels through differentiation. This new development takes place in TAPENADE kernel. As such, although not absolutely necessary for Fortran, it

benefits to differentiated programs in Fortran too. Things may prove harder in Fortran, due to strange situations coming from scattered declaration of a single object among a subroutine and its includes. Think of a `implicit` declaration in a subroutine header that influences variables declared in some included file. In such weird cases, it is sometimes impossible to reuse the original include file. The fallback strategy is then to build a new include file. More generally, we consider the partial order that links all declarations, original and differentiated. When the order allows for it, we prefer to generate an include of the original include file followed by an include of a differentiated include file. Otherwise, our fallback strategy is to put all declarations, original and differentiated, into the differentiated include file. This strategy can be compared to what we did for differentiated modules in Fortran95, which need to keep a copy of the original module's components.

When a program uses external subroutines, TAPENADE expects the user to give some information on these externals via some "black-box" mechanism. In C, the "forward declaration" constraint makes sure that any external subroutine is declared with its arguments number and type before it is used. A similar mechanism exists in Fortran95 with the interface declaration, but it is not compulsory. These forward declarations ease the burden of the "black-box" mechanism. However information on Use/Def, Outputs/Inputs dependencies, and provided partial derivatives is still required. TAPENADE lets the user do so through an ad-hoc file, although an alternative mechanism based on dummy procedures might work just as well.

4 Parameter-Passing Mechanism

The assumptions of the old TAPENADE regarding parameter-passing were inspired solely from Fortran. In Fortran, call by value-result is generally used for values such as scalars that fit into registers and call by reference is generally used for arrays and structures [**?**].

In C, call by value is the only parameter-passing mechanism. One emulates a call by reference, i.e. an input/output parameter by passing a pointer to this parameter. This parameter-passing mechanism is central for all data-flow analysis such as Use/Def, Activity, Liveness, and TBR. With call by reference, the output status of a parameter must be propagated back to the actual parameter inside the calling procedure. With call by value, this propagation must be turned off. Consider for instance the following procedure `F` with formal argument `y`, together with a call to `F`:

```
void F(float y) {
  ...
  y = 0.0;
}

F(x) ;
```

If `x` has a data-flow property e.g., is active just before the call, then so is `y` at the beginning of `F`. Then `y` becomes passive. However in the call by value case, this property must not be propagated back to `x` upon exit from `F`, and `x` remains active after the call. With call by reference or call by value-result, `x` becomes passive after the call. The parameter-passing mechanism used by the language must be stored as an environment variable of TAPENADE.

Incidentally, this also influences the header of differentiated procedures. In several cases the additional arguments to a differentiated procedure must be output arguments, even when their corresponding non-differentiated argument is just an input. This is commonplace in the

reverse mode. It also occurs when transforming a function into a procedure with an extra argument for the result, which is often necessary during differentiation. While this was all too easy in Fortran, now in a call by value context we must pass a pointer to these extra arguments in order to get a result upon procedure exit.

5 Alias Analysis

The data-flow analysis in TAPENADE already dealt with pointers for Fortran95. However, only with C do pointers occur with their full flexibility. Therefore the Internal Language IL that TAPENADE uses as a common representation of any source program, handles pointers with notations and constructs that are basically those of C specifically, `malloc`, `free`, the address-of operator `&`, and the dereference operator `*`. Actually it's the Fortran side that need be adapted: at a very early stage during the analysis, typically during type-checking, each use of a variable which turns out to be a pointer is explicitly transformed into an explicit address-of or dereference operator whenever required. Conversely, it's only in the Fortran back-end that address-of and pointer-to operations are removed, re-introducing the Fortran pointer assignment notation "`=>`" to lift ambiguities.

The principal tool for handling pointers is the *Alias Analysis*, which finds out the set of possible destinations for each pointer for each location in the program. Like most static data-flow analysis, Alias Analysis must make some (conservative) approximations. In particular one must choose to what extent the analysis is *flow sensitive*, i.e. how the order of statements influences the analysis output, and to what extent it is *context sensitive*, i.e. how the various subroutine call contexts are taken into account. Specifically for Alias Analysis, most implementations we have heard of are flow insensitive, and partly context sensitive.

In TAPENADE, we made the choice of a *flow sensitive* and *context sensitive* analysis. By *context sensitive* we mean that this interprocedural analysis considers only realizable call-return paths. However, the called procedure in analyzed only once, in the envelope context of all possible call sites. We made the same choice for the other data-flow analysis such as In-Out or Activity, and we are satisfied with this trade-off between complexity of the analysis and accuracy of the results. Our strategy splits the Alias Analysis in two phases:

- The first phase is bottom-up on the call graph, and it computes what we call *Pointer Effects*, which are *relative*. For instance the *Pointer Effect* of a procedure tells the destinations of the pointers upon procedure's exit, possibly with respect to their destinations upon procedure's entry. In other words at the exit point of a procedure, each pointer may point to a collection of places that can be plain variables, or `NULL`, or destinations of some pointers upon procedure's entry. A *Pointer Effect* can be computed for every fragment of the program, provided it is a standalone flow graph with a unique initial point and a unique end point. Figure 2 shows two examples of *Pointer Effects*. Computing the *Pointer Effect* of a procedure only requires the *Pointer Effects* of the procedures recursively called, and is therefore context-free.
- The second phase is top-down on the call graph, and it computes what we call *Pointer Destinations*, which are *absolute*. At any location in the program, the *Pointer Destination* tells the possible destinations of each pointer, which can be any collection of variables in the program plus `NULL`. This information is self-contained and does not refer to pointer destinations at other instants. On a given procedure, the *Pointer Destinations* analysis collects the contexts from every call site, builds a *Pointer Destinations* and propagates it through the flow graph. When the analysis runs across a call, it does not go inside the

called procedure. Instead, it uses the *Pointer Effect* of this procedure to build the new *Pointer Destinations* after the call. These *Pointer Destinations* are the final result of alias analysis, that will be used during the rest of differentiation.

When the program is recursive, each phase may consist of several sweeps until a fixed point is reached. Otherwise, only one sweep per phase is enough, and the overall complexity remains reasonable.

Pointer Effects and *Pointer Destinations* are represented and stored as matrices of Booleans, using bitsets. For both, the number of rows is the number of visible pointer variables. For *Pointer Destinations*, there is one column for each visible variable that can be pointed to, plus one column for `NULL`. In addition to this, for *Pointer Effects*, there is one extra column for each visible pointer. A *True* element in these extra columns means that the "row" pointer may point to whatever the "column" pointer pointed to at the initial point.

At the level of each procedure, Alias Analysis consists of a forward propagation across the procedure's flow graph. When the flow graph has cycles, the propagation consists of several forward sweeps on the flow graph until a fixed point is reached. Otherwise only one sweep is enough. The propagation is very similar for the first and second phases. Each basic block is initialized with its local *Pointer Effect*. The entry block receives the information to propagate: during the first, bottom-up phase, this is an "identity" *Pointer Effect*, each pointer pointing to its own initial destination. During the second, top-down phase, this is the envelope of the *Pointer Destinations* of all call sites. Actual propagation is based on a fundamental composition rule that combines the pointer information at the beginning of any basic block with the *Pointer Effect* of this basic block, yielding the pointer information at the end of this basic block. At the end of the top-down phase, each instruction is labeled with a compact form of its final *Pointer Destinations*.

We thus need only two composition rules for propagation:

$$\textit{Pointer Effect} \otimes \textit{Pointer Effect} \rightarrow \textit{Pointer Effect}$$
$$\textit{Pointer Destinations} \otimes \textit{Pointer Effect} \rightarrow \textit{Pointer Destinations}$$

Let's give an example of the first composition, which is used by the first phase. The second composition is only simpler. Consider the following code

```
void foo(float *p1, float *q1, float v)
{
  float **p2, *r1 ;
  r1 = &v ;
  p2 = &q1 ;                                   Part A
  if (...) {
    p2 = &p1 ;
  }
  *p2 = r1 ;                                   Part B
  p2 = NULL ;
  ...
```

in which we have framed two parts *A* and *B*. Part *A* starts at the subroutine's entry. Suppose that the analysis has so far propagated the *Pointer Effect* at the end of *A*, relative to the entry. This *Pointer Effect* is shown on the left of Fig. 2. Notice that `r1` (*resp.* `p2`) points no longer to its initial destination upon procedure entry, because is has certainly been redirected to `v` (*resp.* `q1` or `p1`) inside `A`. Part *B* is a plain basic block, and its *Pointer Effect* has been precomputed and stored. It is shown on the right of Fig. 2, and expresses the fact that pointers `p2`

and `*p2` have both been redirected, while the other pointers are not modified. The next step

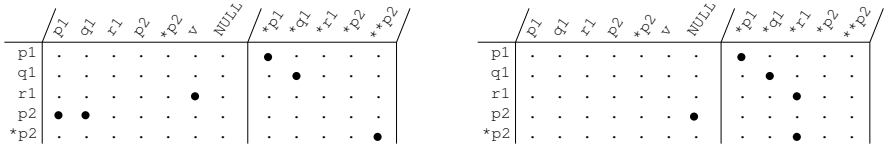
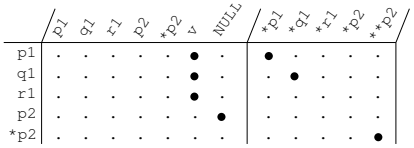


Fig. 2. *Pointer Effects* for part *A* (left) and part *B* (right)

in the analysis is to find out the *Pointer Effect* between subroutine entry and `B`'s exit point. This is done by composing the two *Pointer Effects* of Fig. 2, which turns out slightly more complex than say, ordinary dependence analysis. This is due to possible pointers to pointers. For instance the pointer effect of part *B* states that the destination of `p2`, whatever it is, now points to the address contained in `r1`. Only when we combine with the pointer effect of part *A* can we actually know that `p2` may point to `p1` or `q1`, and that `r1` points to `v`. It follows that both `p1` and `q1` may point to `r1` in the combined result. The *Pointer Effect* for the part $(A;B)$ is therefore:



Although still approximate, these pointer destinations are more accurate than those returned by a flow insensitive algorithm. Figure 3 is a minimal example to illustrate our flow sensitive Alias Analysis (as well as regeneration of declarations and `include` files discussed in Sect. 3). A flow-insensitive Alias Analysis would tell that pointer `p` may point to both `x` and `y`, so that statement `*p = sin(a)` makes `x` and `y` active. Therefore the differentiated last statement would become heavier:

```
bd = bd + (*pd)*y + (*p)*yd;
```

6 Conclusion

The new TAPENADE is now able to differentiate C source. Although this required a fair amount of work, this paper shows how the language-independent internal representation of programs inside TAPENADE has greatly reduced the development cost. Less that one tenth of the TAPENADE has required modifications. The rest, including the majority of data-flow analysis and the differentiation engines, did not need any significant modification.

In its present state, TAPENADE covers all the C features, although this sort of assertion always needs to be precised further. Obviously there are a number of corrections yet to be made, and this will improve with usage. This is especially obvious with the parser, that still rejects several examples. Such a tool is never finished. To put it differently, there are no C constructs that we know of and that TAPENADE does not cover.

Most of the developments done represent either new functionality that may progressively percolate into Fortran too, in the same way that pointers did. Other developments were mostly

Original Code

```
#include <math.h>

void test(float a, float *b)

{
#include "locals.h"


  *b = a;
  /* pointer p is local */
  float *p;

  if (*b > 0) {

    p = &y;
  } else {

    p = &x;
    /*p doesn't point to y*/

    *p = sin(a);
  }
  /* y is never active */

  *b = *b + (*p)*y;
}
```

Tangent Differentiated Code

```
#include <math.h>

void test_d(float a, float ad,
            float *b, float *bd)
{
#include "locals.h"
#include "locals_d.h"
    *bd = ad;
    *b = a;
    /* pointer p is local */
    float *p;
    float *pd;
    if (*b > 0) {
        pd = &yd;
        p = &y;
    } else {
        pd = &xd;
        p = &x;
        /*p doesn't point to y*/
        *pd = ad*cos(a);
        *p = sin(a);
    }
    /* y is never active */
    *bd = *bd + (*pd)*y;
    *b = *b + (*p)*y;
}
```

Original Include File `locals.h`

```
float x = 2.0;
float y = 1.0+x;
```

Generated Include File `locals_d.h`

```
float xd = 0.0;
float yd = 0.0;
```

Fig. 3. *Tangent differentiation of a C procedure. Include directives are restored in the differentiated file. Flow-sensitive Alias Analysis allows* TAPENADE *to find out that* `y` *is not active*

missing parts or misconceptions that the application to C have put into light. But indeed very little has been done that is purely specific to C. In other words, adapting TAPENADE for C has improved TAPENADE for Fortran.

Obviously the main interest of the structure of TAPENADE is that it remains a single tool, for both Fortran and C. Any improvement now impacts differentiation of the two languages at virtually no cost. Even the remaining limitations of TAPENADE, for example the differentiation of dynamic memory primitives in reverse mode, or a native handling of parallel communications primitives, apply equally to Fortran and C. In other words, there is no difference in the differentiation functionalities covered by TAPENADE, whether for Fortran or C. The same

holds probably for the performance of differentiated code, although we have no measurements yet.

There remains certainly a fair amount of work to make TAPENADE more robust for C. However, this development clears the way towards the next frontier for AD tools namely, differentiating Object-Oriented languages. There is already a notion of module for Fortran95, but we foresee serious development in the type-checker to handle virtual methods, as well as problems related to the systematic use of dynamic allocation of objects.

All practical information on TAPENADE, its User's Guide and FAQ, an on-line differentiator, and a copy ready for downloading can all be found on our web address `http://www.inria.fr/tropics`.