# New type signatures for legacy Fortran subroutines*

Nicky WILLIAMS PRESTON
LSV (Laboratoire Spécification et Vérification) URA 2236 CNRS,
Ecole Normale Supérieure de Cachan,
61 Av. Pdt Wilson,94235 CACHAN Cedex,France
Nicky.Williams-Preston@lsv.ens-cachan.fr

**Abstract**

We are currently developing a methodological framework for reverse engineering Fortran77 programs used by Electricité de France, in which the first step is the construction of an algebraic specification which faithfully represents the Fortran code. To construct this specification, we must decide on a coherent set of "profiles" (type signatures) for the specifications of the Fortran sub-programs. We propose an analysis of the dynamic aliases between formal and actual sub-program arguments in order to derive these profiles. In many examples of real Fortran code this analysis does not give satisfactory results if arrays are treated as indivisible. Instead, we must take into account which fragment of the array may really be accessed by the sub-program. We have therefore implemented our analysis as an extension of the *PIPS* code parallelisation tool, which provides us with a precise analysis of inter-procedural array data-flow.
Keywords: reverse engineering, Fortran, alias analysis, dynamic alias, array region

## 1 Introduction

Much of the software currently in use at Electricité de France (*EDF*), though giving entirely satisfactory results, was written 15 or 20 years ago. The replacement of these existing systems by new software re-developed from scratch can be expensive, risky and time-consuming. As a response to this problem, we are developing a methodological framework for reverse engineering Fortran77 source code using formal methods [10], in which algebraic specifications of the code at different levels of abstraction are created and compared.

We propose to start with an analysis of the source code and the creation of the specification at the lowest level of abstraction. We believe that this step can only be partially automated so this first specification must be sufficiently close to the source code to inspire complete confidence that it specifies it exactly. In particular, each sub-program in the code must be specified as a function. However, the specification must not be so low-level as to prevent it being structured and presented so that it is easier to understand than the code itself and so can aid a maintenance engineer in understanding the code and in locating the implementation of particular aspects.

The algebraic specification of a function is called an *operation*. Each *operation* has a *profile* declaring the *sort* of each argument and of the result, i.e. the *profile* specifies the type signature of the function. In order to construct an algebraic specification of the Fortran code, we must decide on the *profiles* of the *operations* which specify the Fortran sub-programs. This is the task which we address in this article.

Several characteristics of the Fortran77 language complicate this task. The declaration of a Fortran sub-program does not tell us which of the declared parameters, nor which of the global variables (i.e. variables in a `COMMON` block) declared as visible, are really *defined* or *used*.[1] Fortran's call-by-address semantics means that passing a single array element to a sub-program can enable it to access the rest of the array from this element onwards. If the passed element is not the first in the array, only a part of the array can be passed to the sub-program. This situation does arise in Fortran77 programs, which make extensive use of arrays, which are the only built-in data-structure,[2] to simulate other data-structures. Fortran77 has few built-in types and does not support user-defined types and, as type-corrrectness is not enforced at compilation, a program may not even respect the declared types.

This means that if we base the *profiles* used in our specification on the declared sub-program parameters and their types, not only do we risk not identifying the true parameters and the true types, but also we are restricted, as in Fortran, to a small set of types. Above all, in the case of sub-programs which are not library routines but belong to a particular application program, we lose the information which can be gleaned from the analysis of the variables actually used to instantiate the formal parameters.

Indeed, the result of Fortran's impoverished type system is that a programmer typically constructs sub-programs with parameters of a certain declared type (`INTEGER`, say), but really only designed to be instantiated by a subset of the program variables of this type, e.g. only the variables numbering lines in an electrical network and not those number-

* This work was financed by Electricité de France through the Contrat d'étude R32/1K2924/ER292, No. OT: R32L02

Proceedings ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99), Toulouse, France, Sep. 1999.

[1] *defined* and *used* are used here in the static analysis sense, meaning respectively assignment of a new value to the variable and reading the value of a variable. In the rest of this paper, we assume familiarity with the notions of static analysis: if not, see chapter 10 of [1]

[2] Apart from chains of characters in some versions.

ing nodes, for example. When the reverse engineer reads the code for the first time, it may not be clear what the parameter represents, particularly since Fortran variable names are limited to six characters.

In this situation, we would like to be able to "track" the values which this parameter may take so as to find which other sub-programs they may be passed to (via variables of different names) and build up an understanding, not necessarily of which values are involved, but of what they represent. In other words, using "type variables" (arbitrary names) because we don't know "what" the types are, we would like to start by finding a consistent set of type signatures for the sub-programs, which are finer-grained than those based on the declared types. These type signatures are just the *profiles* of the *operations* specifying the sub-programs. The specification *sorts* are names for the types we detect in this way, and which we hope correspond more closely than the declared types to the programmer's conceptual view of the code.

Below, we propose a static analysis of the Fortran code in order to automatically obtain a precise definition of the variables which are really *defined* or *used* by each sub-program, and thus to group these variables into sets corresponding to *sorts*. For each set of Fortran variables created, we define a sort which contains at least all possible values of all these variables. We can then define the *profiles* of the *operations* specifying the sub-programs.

In the following section, we discuss our approach further, comparing it to conventional type inference, explaining the hypotheses it is based on and our treatment of array regions using the *PIPS* tool. In Section 3 we give a detailed description of the analysis we propose. Section 4 describes the current implementation and Section 5 discusses a possible extension. Related work is compared to ours in Section 6, and this is followed by the conclusion.

## 2 Further discussion of our approach

### 2.1 Why not conventional type inference?

We want our analysis to distinguish not only integer line-numbers from integer node-numbers, but also the integer arrays used to store linked lists of line numbers from the integer arrays used to store the number of the node connected to one end of each line. Indeed, arrays can vary in the sort of the value stored and the sort of the index, but the values of the elements may also be constrained so that, taken together, they realise a data structure such as a linked list. Moreover, given an array which (in order to simulate dynamic memory allocation) is effectively referenced as though composed of several contiguous sub-arrays, each storing different information, we want to be able to distinguish its different parts. We propose to make these distinctions not by describing exactly how each array fragment is accessed but by putting into the same sort only those array fragments which are accessed in the same way by the same sub-programs.

Classic polymorphic type inference ([7]) is used mostly for functional languages and usually performed on a program already partially typed by means of type declarations. A set of basic types and type-constructors is pre-defined and type variables are assigned to untyped terms. Type inference rules are then used to unify type terms and find consistent types for all the function declarations and function applications in the program. These include those involving primitive operations such as assignment, tests for equality and arithmetic operations. The arithmetic operations are usually given pre-defined non-polymorphic type signatures but functions whose type signature is not known are considered to be polymorphic by default and in particular can take other functions as arguments.

No specific treatment of arrays is proposed, so classic polymorphic type inference is certainly incapable of giving different types to different array fragments.

Moreover, because of the wide use of arithmetic operations, classic polymorphic type inference seems to be incapable of making the distinctions between different integers or different (whole) integer arrays described above. Indeed, in order to avoid a result very close to the declared Fortran types, arithmetic operations and any other "utility" operations on integers and integer arrays must be considered as polymorphic.

On the other hand, sub-sort polymorphism seems sufficient to describe the Fortran programs we have analysed so far. We have found no examples of sub-programs which take other sub-programs as arguments,[3] nor of parametric polymorphism (such as sub-programs which operate on lists regardless of the type of the elements of the list).

Compared to conventional type inference, we have no basic types and propose to ignore type declarations. We only type formal parameters (as defined below) of Fortran sub-programs and only use the equivalent of one of the rules from conventional type inference: the one for function application. Instead of explaining our inference in terms of unification of type terms using this rule, we prefer to describe it as the inference of a relation between parameters of the same type. This enables us to explain how different occurrences of what appears to be the same parameter do not always have the same type.

### 2.2 Our treatment of polymorphism

In contrast to conventional type inference, we propose an analysis which is restricted to analysis of sub-program calls and to the synthesised declarations of these sub-programs. We do not propose to analyse calls to built-in Fortran arithmetic operations, nor tests of equality or assignments.

The analysis of calls to sub-programs which are specific to the management of electrical networks should enable us to make the distinctions between different integer variables or integer array variables described in Section 2.1.

However, "utility" sub-programs may be used to carry out operations on arbitrary integers or arrays. Each *operation* only has one *profile* but, to take such sub-programs into account, we could consider the *sorts* of line-numbers and node-numbers as sub-sorts of the *sort* of all integers and the different *sorts* of integer arrays distinguished above as sub-sorts of the *sort* of all integer arrays. We would then allow implicit coercion between sub-sort and super-sort as necessary. However, this would mean making our analysis (and its results) much more complicated in order to give consistent *profiles* to the routines which are not going to contribute the most to our understanding of the what the variables represent. What is more, it is often the calls to "utility" sub-programs which violate the declared types because they do not in fact embody a sub-sort relation but just a programming "trick" which we have no desire to include in our specification.

In fact, we prefer to limit our analysis to the much less complex case of *profiles* defined over a "flat" set of *sorts*, with

[3] In any case, we would not be able to specify such sub-programs in our first-order algebraic specification language.

no attempt to take a possible sub-sort relation into account and no implicit coercion. The algebraic specification will certainly contain sub-sort relations, but we leave that part of it to the reverse engineer.

We therefore propose to try to recognize the "utility" sub-programs which may be present in the code to be analysed and to "eliminate" them from the code before the analysis is carried out.[4] The choice of sub-programs to eliminate is of course subjective and determines the granularity of the *sorts* which will subsequently be detected. However, in our experience, the utility sub-programs can be easily determined by studying the call-graph of the program, which contains far more calls to these sub-programs than to the other, more specialised ones.

### 2.3 Re-use of variables

Our analysis relies on the hypothesis that successive values of a Fortran variable have the same *sort*. Re-use of variables to save memory is not uncommon in old Fortran code and if successive values are of a completely different nature then we will get bad results. One solution, which we do not have space to discuss further here, is to modify the inference rules (and heuristic) proposed below so as to no longer consider as identical two occurrences of the same region if its value may have been modified between the two. The disadvantage of this solution is obviously that we may unnecessarily give different *sorts* to successive values of variables which in reality do always keep the same *sort*, and so artificially split a lot of variable sets.

### 2.4 Our treatment of arrays

We would like to detect the array fragments which are subjected to a particular treatment by the sub-program being analysed. This arises when a loop in the sub-program traverses the array fragment, even if not every element in the fragment is accessed. It is these array fragments which are used to realise data structures but within the fragment different array elements may be used to represent different things. So instead of giving a sort to each individual array element which may be accessed, or to the whole array (in the case of loops which don't traverse the whole array), we would like to give a sort to the traversed fragment.

The *PIPS* tool performs a static analysis of Fortran code in which sets of accessed elements are described as convex array *regions* [4].

### 2.5 Array *regions* in *PIPS*

*PIPS* is a tool for the automatic parallelisation of Fortran77 programs developed by the Ecole des Mines de Paris [6]. The parallelisation depends on a precise analysis, which can be carried out interactively, of intra- and inter-procedural array data flow. *PIPS* characterises the variables which are really accessed by each block of code and then recursively propagates this information.

A *PIPS* array *region* defines linear limits on the upper and lower values of the index of accessed elements for each dimension of the array. It thus describes the indices of the possibly accessed elements as a convex polyhedron parameterised by scalar program variables.

[4] The elimination of calls to certain sub-programs must be done with care if it is not to invalidate the results of automatic analysis, but this problem is already present in the analysis of programs which may include chains of calls involving thousands of sub-programs unless we can find some way to eliminate those which don't really interest us.

The *PIPS region* is not necessarily an exact description of the set of elements really accessed:

- Obtained by static analysis, it suffers from the limitation of all static analysis, i.e. the inability to eliminate all impossible execution paths.
- The operations used to propagate and combine *regions* for each block of code cannot always give an exact result: a non-convex result may be approximated by a convex polyhedron and non-linear (as a function of scalar variables) limits may be over-approximated.

*PIPS EXACT regions* exactly describe the set of elements accessed either

- in all possible execution paths
- or in a particular execution path for which *PIPS* can express the condition as a linear function of scalar program variables.

Other *regions* may include elements not actually accessed and are called *MAY*.

*PIPS IN regions* describe *upwardly exposed uses* and *OUT regions* describe *definitions* of values which may be *used* on exit from the code block before being re-*defined*.

*PIPS* combines all *IN* or *OUT regions* of the same array for each code block into one *SUMMARY IN* or *OUT region*, respectively.

We will now describe our analysis, using the concept of array regions.

## 3 The analysis we propose

### 3.1 The central rule

The detection of sets of variables corresponding to a *sort* is based on the following simple rule, expressed intuitively:

> all actual parameters instantiating the same formal parameter of the same sub-program $P$ are specified with the same *sort* as that of the formal parameter in the *profile* of the *operation* specifying $P$.

Let us describe how we interpret and apply this rule.

### 3.2 The example

We use as an example the formal parameters called `IPCO` or `IPCE` of certain sub-programs in one of the real programs we have analysed. The call-graph of just these sub-programs, annotated with extracts of the Fortran sub-program declarations and calls, is shown in Figure 1.

### 3.3 Formal parameter regions

Borrowing the *PIPS* terminology, we will call access in *in* mode of a variable by a sub-program any *may* or *must use* which may be executed before any *definitions* of the same variable by the same sub-program. We call access in *out* mode any *definition* by the sub-program which may subsequently be *used* outside the sub-program before being over-written.

We define the *true output parameters* of a sub-program to be

- all scalar variables which may be accessed in *out* mode

Figure 1: Annotated call-graph for the sub-programs in our example

```
SUB. BESTI1

  CALL PREFIX(...,NA1,...)
  SUB. PREFIX(...,NA1,...)

    CALL PRXPLO(NA1(I_IPCO),NA1(I_IPCE),NA1(I_ICCO),NA1(I_ICCE),...,NA1(I_NRLO),NA1(I_NRLE),...)
    SUB. PRXPLO(IPCO,       IPCE,       ICCO,       ICCE,       ...,NRLO,       NRLE,       ...)

      CALL PRCHEN(...,IPCO,ICCO,NRLE)
      CALL PRCHEN(...,IPCE,ICCE,NRLO)
      SUB. PRCHEN(...,IPCO,ICCO,NRLE)

  CALL CAMAT(...,NA1,...)
  CALL CAMAT(...,NA1,...)
  SUB. CAMAT(...,NA1,...)

    CALL CACONT(...,NA1,NA1(I_NRLO),NA1(I_NRLE),...)
    SUB. CACONT(...,NA1,NRLO,       NRLE,       ...)

      CALL CASDEG(...,NA1(I_IPCO),NA1(I_ICCO),NRLE)
      CALL CASDEG(...,NA1(I_IPCE),NA1(I_ICCE),NRLO)
      SUB. CASDEG(...,IPCO,       ICCO,       NRLE)
```

Figure 2: Formal parameter regions for our example

```
<BESTI1:NA1(PHI1)-IN-AP-{19+2BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>
<BESTI1:NA1(PHI1)-OUT-EX-{19<=PHI1, PHI1<=18+BESTI1:F_NS, PHI1<=7118}>
<BESTI1:NA1(PHI1)-OUT-AP-{19<=PHI1, PHI1<=7118}>
<BESTI1:NA1(PHI1)-OUT-EX-{19+BESTI1:F_NS<=PHI1, PHI1<=18+2BESTI1:F_NS, PHI1<=7118}>
<BESTI1:NA1(PHI1)-OUT-AP-{19+BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>

<PREFIX:NA1(PHI1)-IN-AP-{PREFIX:I_IPCO+PREFIX:NS<=PHI1, 1<=PHI1}>
<PREFIX:NA1(PHI1)-IN-AP-{PREFIX:I_IPCE+PREFIX:NS<=PHI1, 1<=PHI1}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCO+PREFIX:NS}>
<PREFIX:NA1(PHI1)-OUT-AP-{PREFIX:I_IPCO<=PHI1, 1<=PHI1}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCE+PREFIX:NS}>
<PREFIX:NA1(PHI1)-OUT-AP-{PREFIX:I_IPCE<=PHI1, 1<=PHI1}>

<PRXPLO:IPCO(PHI1)-IN-AP-{PRXPLO:NS+1<=PHI1, 1<=PHI1}>
<PRXPLO:IPCE(PHI1)-IN-AP-{PRXPLO:NS+1<=PHI1, 1<=PHI1}>
<PRXPLO:IPCO(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PRXPLO:IPCO(PHI1)-OUT-AP-{1<=PHI1}>
<PRXPLO:IPCE(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PRXPLO:IPCE(PHI1)-OUT-AP-{1<=PHI1}>

<PRCHEN:IPCO(PHI1)-IN-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>
<PRCHEN:IPCO(PHI1)-OUT-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>

<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCO+CAMAT:NSO}>
<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCE+CAMAT:NSO}>
<CAMAT:NA1(PHI1)-IN-AP-{CAMAT:I_IPCO<=PHI1, 1<=PHI1}>
<CAMAT:NA1(PHI1)-IN-AP-{CAMAT:I_IPCE<=PHI1, 1<=PHI1}>

<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCO+CACONT:NSO}>
<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCE+CACONT:NSO}>

<CASDEG:IPCO(PHI1)-IN-EX-{1<=PHI1, PHI1<=CASDEG:NSO}>
```

- all array elements which may be accessed in *out* mode by an instruction which is not in a loop, or which is in a loop but uses an array index value which doesn't vary in successive loop iterations
- all array fragments which are *true output parameters* of a sub-program called in an instruction not in a loop, or in a loop in which the array fragment doesn't vary in successive loop iterations, and
- all array fragments which may be *traversed* in *out* mode by a loop, i.e. for each access of an array element or fragment in a loop and which varies with successive loop iterations, the convex envelope of all the elements of the array which may be referenced in *out* mode. In the case of nested loops, we take the fragment traversed by the outermost loop.

The *true input parameters* have a similar definition, except that each array fragment which is a *true output parameter* but in which every element may not be *defined* by the sub-program, and which is not already a *true input parameter*, is added to the *true input parameters*.

We define *in-exact*, *out-exact*, *in-approx* and *out-approx regions* respectively as being the results of the *PIPS* analysis modified so as

- not to calculate *SUMMARY regions*, but one separate region for each array access in the code block, but nonetheless integrating the regions of individual loop iterations of the same access into one region for the whole loop and
- to distinguish between *regions* with exact limits (which we will call *exact*) and *regions* for which the limits have been over-approximated (which we will call *approx*).

Extending the notion of array region to scalar variables (regions with 0 dimensions), we define the *regions of the output parameters* to be all the *out-exact* and *out-approx* regions for the sub-program and the *regions of the input parameters* to be all the *in-exact* and *in-approx* regions, after the addition, if necessary, of a copy of any *may* or non-convex *out* regions to the *in* regions.

In the case of several identical *exact* parameter regions, we only keep one because each is an exact description of the same true parameter. However, in the case of several identical *approx* regions we keep all the copies because the same *approx* region may describe two different array fragments and so two different true parameters. Indeed, some parameter regions may describe fragments which are in reality empty, but which the static analysis cannot identify as such, i.e. true parameters which do not really exist.

The true parameters include (fragments of) variables which are not declared as parameters of the sub-program but just in a `COMMON` block: we will call these *global parameters*.

Our analysis is based on the instantiation of formal parameters by actual parameters but the *global parameters* are not instantiated. These variables may be named differently in different sub-programs but this is an instance of *static aliasing* (like the use of `EQUIVALENCE` statements). A global variable accessed in a sub-program but not declared as an argument may not even have a name (i.e. be declared) in the caller. Note that this is not to say that we do not use global variables in our analysis: a *global parameter* may be used as an actual parameter in a call to another sub-program. Moreover, after our analysis has been carried out any variable which is a static alias of an analysed variable (and thus has the same *sort*) can easily be detected and added to the same variable set.

The *true formal parameters* are the *true parameters* less the *global parameters* and the *formal parameter regions* are defined accordingly.

The formal parameter regions for our example are shown in Figure 2. Each region is enclosed by $<$ and $>$ and contains the name of the sub-program, the name of the array and, in brackets, the values of the indices of the different dimensions represented by `PHI1`, `PHI2`, .... Next comes the mode (*in* or *out*), then the precision of the limits (*exact* or *approx*). Finally, between braces, there is the list of constraints on the index values.

The *in-approx* regions of `BESTI1`, `PREFIX` and `PRXPLO` only exist if `PRCHEN` may read values of `IPCO` and `IPCE` which have not previously been written by `PRXPLO`. *PIPS* is unable to determine if this is the case.

### 3.4 Actual parameter regions

Given a sub-program, $P2$, with one true formal parameter described by the region $f$, and a call-site of $P2$ in sub-program $P1$, the *actual parameter region* for this call-site is the translation of $f$ into the *namespace* (set of all declared parameters, variables of declared `COMMON` blocks and local variables) of $P1$, as carried out by *PIPS*.

In the case of arrays, the array which the formal parameter belongs to and that of the actual parameter, may be declared differently (different dimensions or even Fortran types). As a consequence, a region which is *exact* when expressed in terms of the indices of the formal array may be *approx* when it is expressed in terms of the indices of the actual array [3].

A region is parameterised by the *store* (values of the variables) at the start of the block of code for which it is calculated. However, the values of some variables which parameterise regions may change during the execution of the sub-program. We can only compare regions which are parameterised by the same store. We therefore define the *actual parameter region at the entry-point* as the *actual parameter region* for a call to sub-program $P2$ from $P1$ (which is expressed in terms of the store at the call-site) re-expressed by *PIPS* in terms of the store at the entry-point of $P1$.

If the call-site is in a loop and the actual parameter varies in the different iterations of the loop then it is not the actual parameter for an arbitrary iteration of the loop that is described by a region parameterised by the store at the entry-point. In this case, *PIPS* calculates a new region for the loop which **contains all** the actual parameter regions for the different iterations and it is this region which is re-expressed in terms of the store at the entry-point of the caller.

Also, *PIPS* is sometimes unable to re-express *exact* regions in terms of the store at the entry-point and so has to approximate them by *approx* regions.

### 3.5 Instantiations and the *dynamic_alias* relation

Let us call *instantiations* all pairs
$<$*formal parameter region*,
*actual parameter region at the entry-point*$>$
and define a binary relation on regions *instantiated_by* which links all pairs of regions in an *instantiation*.

Two variables are generally said to be dynamic aliases if they share the same address in memory by virtue of a call to a sub-program. This means that dynamic aliases are just different possible names for the "same" variable. We define another binary relation on regions, *dynamic_alias*, which links regions simultaneously having the same address-space in memory if a particular path in the call-graph is followed.

The *dynamic_alias* relation includes *instantiated_by*. Moreover, if there is an instantiation $< r1, r2 >$ for a call-site, in sub-program $P2$, of sub-program $P3$ and if $P2$ is called by $P1$ then $r2$ may also be the formal parameter region in an instantiation for the call of $P2$ by $P1$. The *dynamic_alias* relation is in fact the transitive closure of *instantiated_by*, i.e.:[5]

$$instantiated\_by(f,a) \Rightarrow dynamic\_alias(f,a)$$

$$dynamic\_alias(r1,r2) \wedge dynamic\_alias(r2,r3) \\ \Rightarrow dynamic\_alias(r1,r3)$$

Neither *instantiated_by* nor *dynamic_alias* are symmetrical.

Let us call *alias list* a list containing the greatest possible number of regions such that each element (except the first) instantiates the one before it and each element (except the last) is instantiated by the one after it. At the point in the execution of the program when the first element in the list is instantiated, i.e. when the sub-program which is deepest in the call-graph is called, all the elements in an alias list simultaneously share the same space in memory and it is then a list of different names for the "same" variable.

The alias lists for our example are shown in Figure 3. If the last actual parameter region in a list is not also a formal parameter region (i.e. is neither *in* nor *out* for the caller), then it is shown in this figure with no mode.

The calls to `PRCHEN` are in a loop and the actual parameter regions cannot be expressed in terms of the store at the entry-point of `PRXPLO`, so there are two alias lists of only one element. In fact, it is the regions of `PRCHEN` which give rise to the *approx* regions of `PRXPLO`.

### 3.6 The *same_sort* relation

We now consider how we can group variables into a set for each *sort*. We define a binary and symmetric relation *same_sort* between regions. This relation *includes* the *dynamic_alias* relation because two regions in the *dynamic_alias* relation are two possible names for the "same" variable and (as we do not consider the possibility of implicit coercions) must therefore be put into the same set. Any pair of elements from the same *alias list* is thus in the *same_sort* relation.

If a sub-program with formal parameter region $f$ is called from two call-sites then each of these call-sites will result in a different alias list (identical up to $f$ and different thereafter), but $f$ will belong to both these alias lists. However, from our initial rule from Section 3.1, we can say that the elements of both alias lists have the same sort. In other words, *same_sort* is transitive "in the direction of *dynamic_alias*":

$$same\_sort(f,a1) \wedge dynamic\_alias(f,a2) \\ \Rightarrow same\_sort(a1,a2)$$

In our example, this rule gives the same sort to the elements of the two alias lists starting with a region of `CASDEG: IPCO`.

Moreover, if the same actual parameter, precisely described by an *exact* region, instantiates two different formal parameters then the actual parameter region will belong to two alias lists which start differently but are identical from this region onwards. The parameter described by this region shares the same space in memory as the variables from both alias lists, but at successive or alternative execution points. We suppose that the value stored at this address always has the same sort, and infer that the elements of both alias lists all have the same sort. In the case of *exact* regions, *same_sort* is thus transitive "in both directions":

$$exact(a) \wedge same\_sort(f1,a) \wedge dynamic\_alias(f2,a) \\ \Rightarrow same\_sort(f1,f2)$$

In our example, this rule gives the same sort to the elements of each pair of alias lists ending in the same *exact* region of `BESTI1:NA1`.

In fact, we can make the same inference as above irrespective of the mode of the regions concerned, so let us make *same_sort* reflexive for *exact* regions irrespective of their mode :

$$exact(r1) \wedge exact(r2) \wedge same\_limits(r1,r2) \\ \Rightarrow same\_sort(r,r)$$

and *same_sort* is an equivalence relation for *exact* regions.

However, if the same *approx* region instantiates different formal parameter regions, then we cannot be sure that the array fragment described by the *approx* region is the same in each case, so we cannot make the same inference as for *exact* regions. *same_sort* is not transitive in this case because although each parameter only has one sort, an *approx* region can have two different sorts because it can describe two different parameters.

We now create sets containing the greatest possible number of regions in which each region is the *same_sort* as all the other elements of the set. For *exact* regions, the sets are the equivalence classes of *same_sort*. Each set of regions represents a sort. The sub-program *profiles* are constructed by giving each set a name and then for each sub-program finding which set each *formal parameter region* belongs to. In the case of several identical *approx formal parameter regions*, each instance of the region should belong to a different set.

The sorts for our example are shown in Figure 4.

### 3.7 Overlapping regions

What inference can we make about the sorts of two regions which overlap to some extent? The situation is resumed in the table shown in Figure 5, which shows all possible ways in which two regions can overlap.

In the case of inclusion of one region in another *approx* region we cannot make any inference because there may not really be inclusion of the array fragments described. The same is true of partial overlap if one region is *approx*.

However, in the case of inclusion of one region in another *exact* one, we know that there really is inclusion of the parameters. In this case, the most obvious explanation for proper inclusion is that the smaller region belongs to a sub-sort of the larger one. However, taking sub-sort relations into account in general would significantly complicate our analysis, and its results. We could decide to incorporate the sub-sort into the super-sort in this case, and give both regions the same sort. However, in the only example of this case that we have seen in the programs we have analysed (and that is not due to a "utility" routine, see Section 2.2),

[5]In this and the following first-order logic formulae, we suppose universal quantification of all variables.

Figure 3: Alias lists for our example

```
<PRXPLO:IPCO(PHI1)-IN-AP-{PRXPLO:NS+1<=PHI1, 1<=PHI1}>
<PREFIX:NA1(PHI1)-IN-AP-{PREFIX:I_IPCO+PREFIX:NS<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-AP-{19+BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>

<PRXPLO:IPCE(PHI1)-IN-AP-{PRXPLO:NS+1<=PHI1, 1<=PHI1}>
<PREFIX:NA1(PHI1)-IN-AP-{PREFIX:I_IPCE+PREFIX:NS<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-IN-AP-{19+2BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>

<PRXPLO:IPCO(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCO+PREFIX:NS}>
<BESTI1:NA1(PHI1)-OUT-EX-{19<=PHI1, PHI1<=18+BESTI1:F_NS, PHI1<=7118}>

<PRXPLO:IPCE(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCE+PREFIX:NS}>
<BESTI1:NA1(PHI1)-OUT-EX-{19+BESTI1:F_NS<=PHI1, PHI1<=18+2BESTI1:F_NS, PHI1<=7118}>

<PRXPLO:IPCO(PHI1)-OUT-AP-{1<=PHI1}>
<PREFIX:NA1(PHI1)-OUT-AP-{PREFIX:I_IPCO<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-OUT-AP-{19<=PHI1, PHI1<=7118}>

<PRXPLO:IPCE(PHI1)-OUT-AP-{1<=PHI1}>
<PREFIX:NA1(PHI1)-OUT-AP-{PREFIX:I_IPCE<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-OUT-AP-{19+BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>

<PRCHEN:IPCO(PHI1)-IN-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>

<PRCHEN:IPCO(PHI1)-OUT-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>

<CASDEG:IPCO(PHI1)-IN-EX-{1<=PHI1, PHI1<=CASDEG:NSO}>
<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCO+CACONT:NSO}>
<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCO+CAMAT:NSO}>
<BESTI1:NA1(PHI1)-EX-{19<=PHI1, PHI1<=18+BESTI1:F_NS, PHI1<=7118}>

<CASDEG:IPCO(PHI1)-IN-EX-{1<=PHI1, PHI1<=CASDEG:NSO}>
<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCE+CACONT:NSO}>
<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCE+CAMAT:NSO}>
<BESTI1:NA1(PHI1)-EX-{19+BESTI1:F_NS<=PHI1, PHI1<=18+2BESTI1:F_NS, PHI1<=7118}>
```

Figure 4: Sorts for our example with "conservative" treatment of *approx* regions

```
<PRXPLO:IPCO(PHI1)-IN-AP-{PRXPLO:NS+1<=PHI1, 1<=PHI1}>
<PREFIX:NA1(PHI1)-IN-AP-{PREFIX:I_IPCO+PREFIX:NS<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-IN-AP-{19+BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>

<PRXPLO:IPCE(PHI1)-IN-AP-{PRXPLO:NS+1<=PHI1, 1<=PHI1}>
<PREFIX:NA1(PHI1)-IN-AP-{PREFIX:I_IPCE+PREFIX:NS<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-IN-AP-{19+2BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>

<PRXPLO:IPCO(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCO+PREFIX:NS}>
<CASDEG:IPCO(PHI1)-IN-EX-{1<=PHI1, PHI1<=CASDEG:NSO}>
<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCO+CACONT:NSO}>
<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCO+CAMAT:NSO}>
<BESTI1:NA1(PHI1)-OUT-EX-{19<=PHI1, PHI1<=18+BESTI1:F_NS, PHI1<=7118}>
<PRXPLO:IPCE(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCE+PREFIX:NS}>
<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCE+CACONT:NSO}>
<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCE+CAMAT:NSO}>
<BESTI1:NA1(PHI1)-OUT-EX-{19+BESTI1:F_NS<=PHI1, PHI1<=18+2BESTI1:F_NS, PHI1<=7118}>

<PRXPLO:IPCO(PHI1)-OUT-AP-{1<=PHI1}>
<PREFIX:NA1(PHI1)-OUT-AP-{PREFIX:I_IPCO<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-OUT-AP-{19<=PHI1, PHI1<=7118}>

<PRXPLO:IPCE(PHI1)-OUT-AP-{1<=PHI1}>
<PREFIX:NA1(PHI1)-OUT-AP-{PREFIX:I_IPCE<=PHI1, 1<=PHI1}>
<BESTI1:NA1(PHI1)-OUT-AP-{19+BESTI1:F_NS<=PHI1, 1<=PHI1, PHI1<=7118}>

<PRCHEN:IPCO(PHI1)-IN-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>
<PRCHEN:IPCO(PHI1)-OUT-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>
```

Figure 5: Relation between the sorts inferred in the case of overlapping regions

| overlap relation * | *exact* * *exact* | *approx* * *exact* | *exact* * *approx* | *approx* * *approx* |
|---|---|---|---|---|
| $=$ | $=$ | $\leq$ | $\leq$ | - |
| $\subset$ | $<$ | $<$ | - | - |
| other | anomaly | - | - | - |

Figure 6: Sorts for our example using the heuristic for *approx* regions

```
<PRXPLO:IPCO(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCO+PREFIX:NS}>
<CASDEG:IPCO(PHI1)-IN-EX-{1<=PHI1, PHI1<=CASDEG:NSO}>
<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCO+CACONT:NSO}>
<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCO<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCO+CAMAT:NSO}>
<BESTI1:NA1(PHI1)-OUT-EX-{19<=PHI1, PHI1<=18+BESTI1:F_NS, PHI1<=7118}>
<PRXPLO:IPCE(PHI1)-OUT-EX-{1<=PHI1, PHI1<=PRXPLO:NS}>
<PREFIX:NA1(PHI1)-OUT-EX-{PREFIX:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=PREFIX:I_IPCE+PREFIX:NS}>
<CACONT:NA1(PHI1)-IN-EX-{CACONT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CACONT:I_IPCE+CACONT:NSO}>
<CAMAT:NA1(PHI1)-IN-EX-{CAMAT:I_IPCE<=PHI1, 1<=PHI1, PHI1+1<=CAMAT:I_IPCE+CAMAT:NSO}>
<BESTI1:NA1(PHI1)-OUT-EX-{19+BESTI1:F_NS<=PHI1, PHI1<=18+2BESTI1:F_NS, PHI1<=7118}>
<CAJAC:IPCO(PHI1)-IN-EX-{1<=PHI1, PHI1<=CAMAT:NSO}>
<CAJAC:IPCE(PHI1)-IN-AP-{1<=PHI1, PHI1<=CAMAT:NSO}>

<PRCHEN:IPCO(PHI1)-IN-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>
<PRCHEN:IPCO(PHI1)-OUT-EX-{PHI1==PRCHEN:N1, 1<=PHI1}>
```

the "super-sort" exists only for purely operational purposes, to save a few lines of code. We prefer therefore to leave the reverse engineer to analyse such cases.

In the case of an *approx* region having exactly the same limits as an *exact* region, we can suppose that the sort of the *approx* region is the same as or a sub-sort of the sort of the *exact* region and proceed as above. Below, we propose an alternative approach for cases such as this.

We cannot explain partial overlap of two *exact* regions, and can only alert the reverse engineer to an apparent anomaly.

## 4 The current implementation

A first implementation of our analysis has been realised as a new *PIPS* phase and run on three real legacy Fortran programs of 1000, 3500 and 25500 lines of code, respectively. The aim of this first tentative implementation was to enable us to quickly validate our ideas on real code and so we based it on the already-calculated *PIPS* regions.

In these trials, we obtained nearly one sort for every two "regions" (including scalars). In the analysed programs, the number of different call-sites for any sub-program that is not a utility routine is small, but this implementation of our analysis nonetheless fails to recognize that many pairs of regions describe the same array fragments. This is because

1. this is a very approximate implementation of the algorithm described above, because the modifications of the *PIPS* analysis described in Section 3.3 have not been carried out and
2. these programs give many *approx* regions and our analysis is extremely conservative in never concluding that an *approx* region describes the same array fragment as another region.

However, the results provided by this implementation enabled us to refine our analysis.

## 5 A less conservative analysis

Treating each *approx* region as different from all other regions for the same array, and with its own sort, gives a result which is very conservative. If the code contains many *approx* regions, the result is an explosion in the number of apparently different sorts.

In the programs we have analysed, many array accesses are made with index values which depend on the value of another array element or are recursively calculated by the loop containing the access. Putting limits on the array fragment in these cases requires a knowledge of the limits of the values of another array and/or inductive reasoning. *PIPS* can only describe such array accesses by *approx* regions.

One approach is to look for heuristics to suggest when two regions describe the same true parameter, even though they are not both *exact*. We can then make the inferences that follow while signalling the need for the reverse engineer to check by hand the hypothesis on which they are based.

For example, it seems reasonable to suppose that the same array fragment may sometimes be described by an *exact* region for one access point and an *approx* region for another. A heuristic which works well for the codes we have analysed is the following:

> if an *exact* region and an *approx* region of the same array have the same lower limit, which is not the lower limit of the array, and the upper limit of the *approx* region is greater than or equal to that of the *exact* region, then we suppose that the two regions describe the same array fragment.

This heuristic has to be applied with some care:

- if there are two different candidate *exact* regions for the same *approx* region then the heuristic cannot be applied;
- *in* and *out* regions are calculated by taking the difference of other regions, but if the calculation of one *approx* parameter region, $r1$, uses another *approx* parameter region, $r2$, then the heuristic must be applied to $r2$ first;
- after deciding where to apply it first, the implications of the heuristic must be propagated back as far as possible and then the *in* and *out* regions recalculated.

The results of applying this heuristic to our example are shown in Figure 6. Note that the the *in-approx* regions of `BESTI1`, `PREFIX` and `PRXPLO` have been eliminated. The reverse engineer must check by hand whether (or under what pre-conditions on input variables) the value of `PRCHEN:N1` is bounded by 1 and `PRXPLO:NS` in the calls from `PRXPLO` and thus whether the loop of `PRXPLO` traverses the first `NS` elements of `IPCO` and `IPCE`.

## 6 Comparison with other work

Related work includes:

- work on alias analysis: although this mostly addresses programs written in C in which array accesses are not a notable feature but it is rather the analysis of pointers which poses a problem, the work of [12] and [9] can be compared to ours;
- type inference in the *Lackwit* system [8].

The aim of the work described in both [12] and [9] is flow-insensitive pointer alias analysis for C programs. [12] consider not only (as we do) different names in different sub-programs for the same memory location or pointer, but also all the different "names" (access paths) which can be obtained in C using pointers, dereferencing and structure field accesses. They define a relation between two "names" which may, because of an assignment of a value to a pointer or a sub-program call involving pointers, refer to the same memory location at some point in the execution. From the transitive, reflexive and symmetric closure of this relation, they then generate an equivalence relation on the "names", which enables them to partition the code studied and decide for each independent code fragment whether to apply a flow-sensitive pointer alias analysis.

[9] uses type inference in a non-standard type system as an efficient way to realise flow-insensitive pointer alias analysis for C programs. In his type system, a "type" represents a set of variable locations: the locations which may all be pointed to by the same pointer. Each "type" has a "component" representing the locations which may be pointed to by the variables of that "type". All pointers to the same location do not have the same "type" but do have the same type "component". The type system thus represents a graph, the *storage shape graph*, and "type" equality is not the standard structural equality (same components) but is the same as would be obtained by *tagging* the types.

[8] apply to C programs an ML-style ([7]) type inference, but in which the types are tagged. The aim is to detect variables having the same "representation" (i.e. they then have the same "type"). Type inference rules give the same "type" to two variables if one is assigned the value of the other or if one instantiates the other in a sub-program call. However, as in ML, a formal parameter may be polymorphic and so have several "types".

We exploit the similarity between alias analysis and type inference, not to use non-standard-type inference for alias analysis, but to use alias analysis to determine non-polymorphic "types" for formal sub-program parameters in order to construct an algebraic specification. Our analysis is specifically adapted to Fortran and based on array regions.

## 7 Conclusion

The motivation for the work described in this paper is the reverse engineering of legacy Fortran77 code. In order to construct an algebraic specification of the Fortran code, we need to determine a consistent set of type signatures for the subprograms, and we cannot rely on the Fortran declarations. We analyse the actual parameters used in sub-program calls to determine a finer-grained type for each formal parameter of each sub-program, based on the operations used to read or write the values it may take. In the case of formal parameters which are array values, we cannot assume that the whole array has the same type. We therefore use static analysis to characterise as an array "region" each array fragment which may really be traversed by the sub-program. We then determine the type of this region. The traversed fragment cannot always be precisely characterised and represented so we must take into account that some regions are over-approximations. We adopt a conservative response to approximate regions which results in many types being detected. To decide which of these types can be merged, we can use a heuristic which can be checked by the user. A first, approximate, version of our analysis has been implemented as an extension to the *PIPS* tool, which offers precise inter-procedural analysis of array data flow.


### Acknowledgements

I would like to thank Jérôme Ryckbosch of *EDF* for giving me a real-life problem to work on and the *PIPS* team at the Ecole des Mines, in particular Béatrice Creusillet, François Irigoin, Fabien Coelho, Corinne Ancourt and Pierre Jouvelot, for all their help. Thanks also to my supervisor, Michel Bidoit, for his patient re-reading of this article.


### References


[1] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Massachusetts, 1986.

[2] See http://www.brics.dk/Projects/CoFI/.

[3] Creusillet, B. *Array Region Analyses and Applications.* PhD thesis, Ecole des Mines de Paris, December 1996. Available at http://www.cri.ensmp.fr/pips/.

[4] Creusillet, B., and Irigoin, F. Interprocedural array region analyses. In *proc. Workshop on Languages and Compilers for Parallel Computing, LCPC'95, Columbus, Ohio, USA, 10-12 August, 1995* (1995). Also available as TR A/270 at http://www.cri.ensmp.fr/pips/.

[5] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing single static assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 4 (1991), 451–490.

[6] Irigoin, F., Jouvelot, P., and Triolet, R. Semantical interprocedural parallelisation: An overview of the PIPS project. In *proc. International Conference on Supercomputing, June 1991* (1991), pp. 144–151. Available at http://www.cri.ensmp.fr/pips/.

[7] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 4 (1978), 348–375.

[8] O'Callahan, R., and Jackson, D. Lackwit: A program understanding tool based in type inference. In *International Conference on Software Engineering, Boston, MA, 1997* (1997), pp. 338–348.

[9] Steensgaard, B. Points-to analysis in almost linear time. In *proc. Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1996* (1996), pp. 32–41. Available at http://www.research.microsoft.com/ rusa/papers.html.

[10] Williams-Preston, N. An experiment in reverse engineering using algebraic specifications. Tech. Rep. 97-10, LSV, Ecole Normale Supérieure de Cachan, 1997. Available at http://www.lsv.ens-cachan.fr/Publis/.

[11] Wirsing, M. Algebraic specification. In *Handbook of Theoretical Computer Science* (BW, 1990), J. van Leeuwen, Ed., Elseiver, pp. 675–788.

[12] Zhang, S., Ryder, B. G., and Landi, W. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering, San Francisco, October, 1996* (1996). Available at http://www.prolangs.rutgers.edu.