File - Proj Guardian

PROJECT GUARDIAN
TECHNICAL COORDINATION LETTER

Date: 27 January 1976

To: Contracting Officer
HQ ESD/MCP
Hanscom Air Force Base
Bedford, Mass. 01731

TCL No: 13

Contract No: F19628-74-C-0193

Attention: C. E. Fenton, Captain, USAF

Subject: PL/I As A System Programming Language For A Certifiable Multics - January 26, 1976

The attached technical note provides the above listed report as prepared by R. Feiertag.

The memo addresses the problem of identifying a suitable programming language for implementing a certificable Multics system. Tables 1, 2, and 3 are provided to list the acceptability of data types, storage classes and statements, respectively.

If there are any questions, please contact the undersigned or Mr. N. Adleman in our Cambridge, Massachusetts office.

Very truly yours,

HONEYWELL INFORMATION SYSTEMS INC.

R L Carlson

R. L. Carlson
Contract Specialist

RLC/meb
Attachment

cc: ESD/MCI (5)
MITRE-D73, Mr. E. Burke (5)
RADC/ISM (3)
NSA/R14 (3)
AFDSC/XMS (2)
JTSA (5)

01/26/76

PL/I AS A SYSTEM PROGRAMMING LANGUAGE FOR A CERTIFIABLE MULTICS

R. Feiertag

This memo addresses the problem of identifying a suitable programming language for implementing a certifiable Multics system. The language must meet the following goals:

1. The language must be suitable for use in the hierarchical structure used in proving that the system is secure. The language must be able to support hierarchical levels of abstraction with interaction between the levels being restricted to procedure and function calls only. The language should support data abstractons, e.g., each level should be able to create data objects that can be passed to higher levels, but can be operated upon only by functions of the creating level.
2. Programs written in the language must be able to be proven consistent with the system specifications, using contemporary program verification techniques.
3. The language must be suitable for implementing programs of an operating system, namely Multics, i.e., it must have language features suitable for system programs.

The PL/I language clearly meets the third goal because the Multics system is already written mostly in PL/I. However, PL/I does not meet either of the first two goals. The remainder of this memo discusses the possibility of modifying the PL/I language so that it can meet the first two goals without sacrificing the third goal. An additional constraint to the modifications of the PL/I language is that the modifications should not require extensive rewriting of existing Multics programs. An alternative to the use of PL/I is to come up with another language, be it an entirely new, existing, or a modified existing language. However, there is no existing language that demonstrably meets all three goals. Developing an entirely new language requires a large effort. Whatever new language might be chosen, the Multics system would have to be entirely recoded. For these reasons, a suitably modified version of PL/I is a reasonable alternative.

In general, the modifications to PL/I involve eliminating some language constructs and restricting some others. The eliminations and restrictions are summarized in Tables 1, 2, and 3. The philosophy of elimination is to include in the modified language only those features which are necessary for Multics system programming. Unnecessary features should be eliminated. Therefore, features such as PL/I I/O and pictures are eliminated. The restrictions on the language constructs will be discussed later in this document.

Expressions

Expressions are acceptable constructs. The following operators may be used in expressions:

+, -, *, /, **, ^, ||, =, ^=, <, ^<, >, ^>, <=, >=, &, |

Conversions

In general, conversions between data types should be allowed. However, as is the current practice, all conversion between data types should be done explicitly using the appropriate built-in functions. This makes building an automatic verifier easier, since the verifier does not have to know the implicit conversion rules. Only conversions defined explicitly by the PL/I language should be permitted. Implementation dependent conversions such as the "unspec" built-in functions and pseudobuilt-in and the mismatched overlaying of based variables should not be allowed. It is very difficult to build a verifier which can place a proper semantic interpretation upon these conversions.

The number of different kinds of conversion between data types should be minimized. Each conversion makes verification slightly more difficult. Very little, if any, type conversion should be necessary in the Multics kernel.

Implicit conversion between data of the same type, but different precision, is allowed. It is simpler than conversion between types and is harder to avoid.

In all arithmetic data, fixed numbers are preferable to floating numbers because fixed values do not have a truncation and round-off problem. It would be nice if float could be eliminated altogether from the kernel, but this is probably not possible.

Pointers

In general, in order to to build a verifier that can properly interpret references using different types of based variables qualified by the same pointer, the verifier must understand the implementation of the language as well as the definition of the language. The necessity of using knowledge of the implementation significantly complicates the verification process, making verification of programs using such constructs nearly impossible with contemporary techniques. To make verification possible, pointers should be typed so that a pointer can qualify only one type of based variable. To accomplish typing of pointers, the following informally stated rules should be added to the language regarding pointers:

1. Any pointer value used as a qualifier in a based reference must have a type, and that type must be identical to the type of the based variable being qualified.
2. The type of a pointer value identified by a pointer variable is the type of the pointer variable.
3. The type of a pointer value returned by the addr built-in

function is the type of the argument to the function.

4. All pointer values must have a type.
5. All pointer variables must have one and only one type.
6. The type of a pointer variable is the type of the based variable whose declaration contains the pointer variable as the object of the "based" clause. Since each pointer variable must have one and only one type, each pointer variable must appear in one and only one based clause. (It is theoretically possible for the same pointer variable to appear in two based clauses of declarations for variables of the same type.)
7. A pointer value of a given type may be assigned only to a pointer variable of the same type.
8. Pointers are primitive data types and cannot be subdivided. Therefore, the special built-in functions which subdivide a pointer (e.g. baseptr, rel, baseno, and ptr) are eliminated. (Unfortunately, this rule contradicts goal number 3 above; this problem will be dealt with below.)
9. The only permissible operation on pointers and pointer variables are assignment, equality, and qualification. Therefore, the addrel built-in function is eliminated.
10. Declaration of parameter types in an entry declaration must include the type of any pointers passed as an argument. For example:

```
declare doall entry (fixed binary (35),
                     pointer (char (32)));
```

declares doall to be an entry with two parameters: the first parameter is an integer and the second a pointer of type character string, i.e., the pointer must point to a character string.

The above rules are useful in that they assure that all primitive data values of the language will be interpreted in a consistent manner. For example, a data value of the integer 5 will always be interpreted as the integer 5 and never as a bit string, character string, or floating point number. However, these rules do not prevent misinterpretation of higher level abstract data values. For example, one such abstract data value might be a directory entry. A directory entry would be represented as a structure of more primitive data values. However, under the above rules, it is possible for a directory entry to be interpreted as something else if that something else happens to have the identical structure consisting of the same primitive data types. Representations for different abstract objects should be considered to be different types even if the structures which are their representations are identical. There is no straightforward way of introducing this distinction into PL/I. One partial solution involves the use of the "like" attribute. Two structures are considered to be of the same type

only if one is declared to be like the other or they are both declared to be like the same structure. For example, consider the following structures:

```
dcl 1 a,
    2 b fixed bin (35),
    2 c bit (36);

dcl 1 d like a;

dcl 1 e like a;

dcl 1 f,
    2 g fixed bin (35),
    2 h bit (36);
```

In this case, a and d would be of the same type, d and e would be of the same type, however f and a would not be the same type and neither would f and d nor f and e. In order for this scheme to work, all abstract objects would have to be represented as structures even if they consist of only a single primitive element. The main difficulty with this scheme is that all structures would be considered as abstract objects, and this is not always desirable. However, the flaw is not fatal.

In order to implement the hierarchical levels of abstraction, it is necessary to allow one level to pass a reference (typed pointer) to an abstract object (represented as a structure) to a higher level and yet not give the higher level the ability to access the representation of the object. For example, page control might create an abstract object which is a page table. It may pass a reference to the page table to the higher level constituting segment control, however, segment control should not directly access the page table. Segment control must call page control, giving the reference to the page table as an argument, in order to gain access to or effect the page table. This type of encapsulation of data and procedures exists in CLU and ALPHARD and should exist in the modified PL/I. Strictly speaking, there should be no legitimate operation which can be performed at the higher level upon such a reference. However, it is reasonable to allow equality and assignment of such references; qualification cannot be allowed at the higher level because that would permit the higher level to access the representation of the abstract object. Constructs to permit identification of such references should be included in the language, i.e., constructs should be created which permit the compiler and verifier to distinguish those procedures which maintain an abstract object and, therefore, may access its representation, from those procedures which see the object in its abstract form only and, therefore, can not directly access its representation.

Flow of control

Two types of constructs cause difficulty in the area of flow of control. These are: (1) labels and gotos and (2) conditions. In verifying a program containing labels and gotos, each label must have assertions associated with it that describe the state of the program for all possible places from which the label can be transferred to. The use of label variables makes the problem much worse since a goto with a label variable could transfer to many different labels. Labels and gotos should, therefore, be used carefully so that the verification of the program does not become overly complex. Legitimate uses of gotos and labels might be as a means of fanning in to common code by a program with many distinct but similar entry points, or to transfer to a common set of code that returns to the program's caller with an error code. However, gotos should not be used to program loops that can be achieved using a do statement.

An even more difficult problem arises when using a goto to a label in a different procedure, i.e., a non-local transfer between procedures. The formalization and verification of such statements must be considered a research problem. This problem can probably be solved in the near future, however, such verification is still likely to be difficult and non-local transfers between programs should be avoided.

Similar problems exist for the condition mechanism. As far as I know, no work has been done on the verification of programs that have a condition mechanism. It appears impractical to eliminate the condition mechanism from the language because it is used significantly by the system. It is, therefore, clear that work will have to be done in finding a way of formalizing and verifying conditions. used carefully.

Protection of levels

In order to maintain the hierarchical system structure, certain limitations should be placed on a given procedure as to what other procedures it can call. Procedures must not call procedures in higher levels. Also procedures must not call hidden procedures in lower levels.

To make it easy to ascertain that only permitted calls are made, each procedure should have associated with it the level of which it is a part. In addition, each entry to a procedure should have associated with it the highest-level procedure which may invoke it. Levels should also be associated with entry values which are passed as parameters or assigned to entry variables. A call is valid only if the level of the calling procedure is greater than or equal to that of the called procedure and less than or equal to that of the called entry.

In addition to protecting interlevel calls, it is necessary to protect interlevel data references. The only legitimate interlevel data references should be through arguments. Any other interlevel data references are illegal. One way of eliminating such interlevel data references is to require that all data declared in programs be declared internal. This

prevents any interprocedure data references except through pointers (the pointer problem has already been dealt with). This solution is acceptable, but is more restrictive than necessary. Two procedures in the same level may legitimately share data. To accomodate this sharing, external data should be permitted, but each item of external data must reside at a level. In order for an external data reference by a procedure to be legitimate, the level of the procedure must be equal to the level of the data.

Note that levels need not necessarily be integers. Names may be used to denote levels as long as there is some ordering relation for the names. Also the level information need not be (and probably should not be) included within the programs themselves. The level information can be kept separate from the programs in a separate document. This is useful because the level information is part of the specifications as well as the implementation programs and can be used to verify both. Also, having the level information in a separate central document makes the structure of the hierarchy easier to modify.

In order to complete the protection of levels, the PL/I language must assure that only legitimate linguistic references are permitted. This means that array references must be within array bounds and string references must be within string bounds. Therefore, array bounds and string bounds must be checked when the arrays and strings are referenced.

Built-in functions

Unless stated below, all the PL/I built-in functions may be used. However, only those functions which are necessary should be retained. This is because each function retained must be formally described to the verifier. If a function is not necessary, there is no reason to have to formally describe it. The built-in function "unspec" and the special pointer functions should be eliminated for reasons stated above. Elimination of the unspec function should not create any serious problem, however, some of the pointer functions are necessary. For example, the purpose of the Multics "initiate" operation is to generate a pointer to a segment. The "initiate" operation requires at least one of the pointer built-in functions in order to generate a pointer. Hopefully, there are very few operations that need know the internal representation of a pointer, and these will have to be verified in a special way. Also, since pictures and PL/I I/O have been eliminated, built-in functions relating to these features (such as valid, onfile, and onkey) are also eliminated.

Finally, a few words on the "defined" attribute. The "defined" attribute permits certain types of overlaying of data structures. In general, overlaying should be strongly discouraged because it is difficult to semantically interpret an overlay. However, the types of overlaying which can be done using the defined attribute are limited, so a verifier might be able to interpret it without excessive difficulty. If the defined attribute is necessary, work will have to be done in

finding a good formal semantic interpretation for it. The ultimate retention of the defined attribute will depend upon whether or not such a good interpretation can be found.

Conclusion

This memo has discussed the problems of modifying the PL/I language so that it is suitable for programming a version of Multics that can be proven secure. Solutions for some of these problems have also been discussed. Although not all of the problems have been resolved, the design and implementation of a modified PL/I to meet the goals stated at the beginning of this memo should be possible. Since the modifications discussed here consist only of restrictions to the language, compliance with the restrictions can be checked by a preprocessor and no changes to the PL/I compiler are necessary. However, it may be simpler to modify the compiler to include the restrictions than to build a suitable preprocessor. A verifier will have to be constructed to demonstrate the consistency of the restricted PL/I programs with the specifications and the level information. Clearly, there are some difficult problems yet to be resolved and many details yet to be worked out.

Table 1

Acceptability of data types

fixed	retained
float	retained if necessary
bit string	retained
char string	retained
offset	eliminated
pointer	restricted
label	restricted
format	eliminated
entry	restricted
file	eliminated
array	restricted
structure	retained

Table 2

Acceptability of storage classes

automatic	retained
static	restricted
controlled	eliminated
based	retained
parameter	retained
defined	retained if necessary

Table 3

Acceptability of statements

allocate	retained	assignment	retained
begin	retained	call	restricted
close	eliminated	declare	retained
default	eliminated	delete	eliminated
do	retained	end	retained
entry	retained	format	eliminated
free	retained	get	eliminated
goto	restricted	if	retained
locate	eliminated	null	retained
on	restricted	open	eliminated
procedure	retained	put	eliminated
read	eliminated	return	retained
revert	restricted	rewrite	eliminated
signal	restricted	write	eliminated