Dan Greening
University of Michigan
May 26, 1980

Type-checking Loader Records

Fred Swartz's ability to come up with incredibly difficult type-representation problems has been invaluable in the development of this paper. Thanks Fred.

During the course of my playful romps with Pascal, PLUS, and FORTRAN, I have noticed an interesting phenomenon. There seem to be a fair number of bugs that are caused by simply not calling a routine correctly. It wants a halfword, and I want to shove a fullword down it's throat. Or I forgot to give it that last parameter that causes the frammis to do what it is supposed to.

When I worked as a counsellor, one of the most common (and extremely hard to find) bugs that people could bring us were those involving mismatched common blocks. One has to sit down with a large piece of code, find all of the same-named common blocks, and make sure that the types of all the identifiers match. This is not easy in itself, but it is made more difficult because some people do not use the same identifier names in similar declarations.

When I worked in FORTRAN, I had type-mismatch problems all the time. Recently, I have been doing almost all of my work in Pascal and PLUS. Many of the problems that once plagued me were diminished by an order of magnitude. One of the reasons is that both Pascal and PLUS require that the parameters of routines match in type (as long as they are compiled together).

However, I still run into a few problems, especially when I am using separately compiled routines. I change the definition for one routine or I add a field to a shared structure, and forget to recompile a routine that uses it. When a bug appears, it is not always clear what caused the error to occur.

The StonyBrook Pascal compiler solves this problem by having its own linkeditor (if you want to use it) that makes sure that all types match. That's OK, but it creates another problem, because you have to run everything through the linkeditor all the time. There is no StonyBrook-equivalent of *OBJUTIL, so one loses some convenience. And anyway, it doesn't work for PLUS or FORTRAN.

The goal of this paper is to explore the possibilities of type-checking using UMICAD, the MTS loader. In this paper, a simple TYP record format is given first. Its capabilities and limitations are discussed. Later, a more complex and general-purpose TYP record format is described. Examples are given for the representation of type-restrictions with this format.

In both instances, loader-algorithms are given in a Pascal/ALGOL68

hybrid pseudo-code. They are not necessarily complete, but they give some flavor of the type-checking operations. The algorithms are not a requirement for the understanding of the operation of these records. The reader is encouraged to skim them and later, if desired, step through them with some of the examples.

In this paper, you will note that an attempt is made to avoid requiring the loader to know anything about types that can be used by specific compilers. In fact, neither of the models require the loader to know anything about type-restriction. Both models can be thought of as simple sets of generalized comparison rules. The second format is just a set of slightly more sophisticated comparison rules than is the first.

The reason that a model was not considered that required the loader to have compiler-specific information, is that it would increase the complexity of the code required to process the information. And, as you will soon see, it is not necessary anyway.

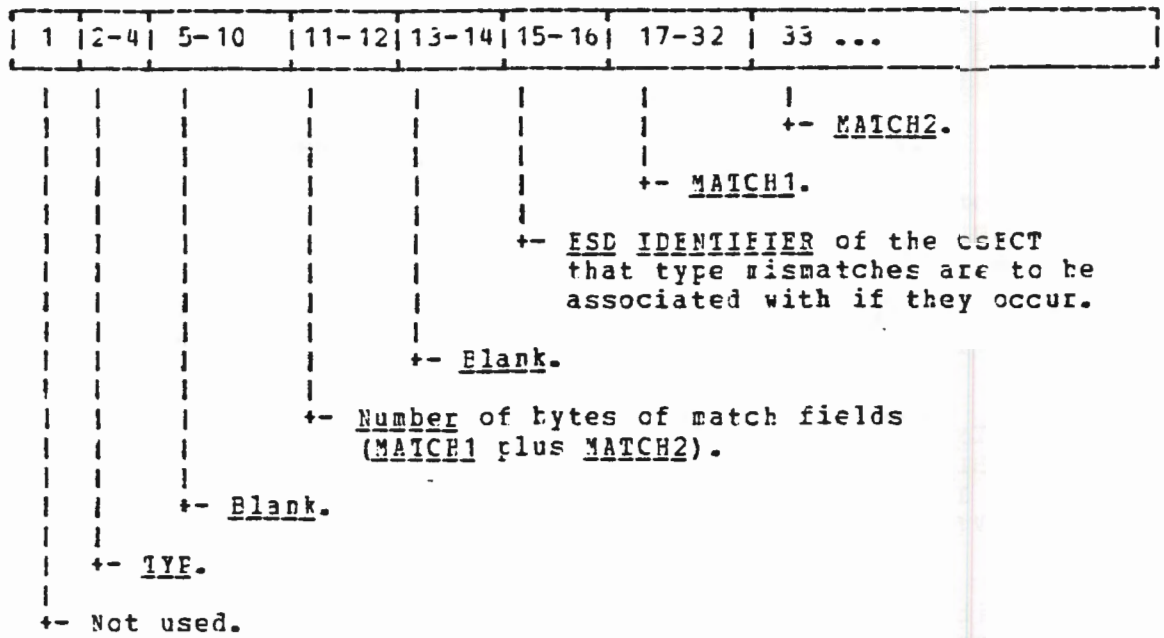
There is an inherent limitation with both of the models that will be discussed. Neither were developed with the idea of checking types between routines that come from different compilers. Using some standard type-restriction representation would merely limit the amount of type-checking that could be done for strongly-typed languages (such as Pascal or PLUS) to the amount of type-checking that can be done for a relatively weakly-typed language (such as FORTRAN).

That is not to say that some standard type-restriction format cannot be used with the records given; however, if this is done it is recommended that compiler-specific type records be produced, too. It will be obvious how that can be accomplished from reading the descriptions.

Another thing that I would like to point out is that I recently talked to Alan Ballard about various other ideas about the loader and he mentioned a long-standing problem with the loader, which is that it can only handle a maximum of 8 characters in an external symbol. We have talked about a number of solutions to that problem, but I did not take them into consideration when writing this paper. Thus, the first MATCH fields in both TYP formats can only accommodate 16 characters. If this is ever implemented, it should take a more modern format than the one given by handling long MATCH fields.

Proposed Format Number I

TYP Input Record



The operation of the loader with respect to this record is as follows:

```
if CPI TYPECHECK=CN
  then
  | Search list for a TYP record with a MATCH1 field
  |     matching this one.
  | If one exists
  |   then
  |   | LEN := minimum of the lengths of the two records.
  |   | Compare the two records with a length of LEN.
  |   | If they are not the same
  |   |   then
  |   |   | Spit out a warning message
  |   |   fi
  |   | If the length of the record we just read is greater than
  |   |     the one we found
  |   |   then
  |   |   | Junk the one we found.
  |   |   | Replace it with the one we just read.
  |   |   fi
  |   else
  |   | Add this record to our TYP record list.
  |   fi
  fi
```

On the compiler side of this, the MATCH1 field would probably be contain the name of the routine or common area (taking 8 bytes), a code name for the compiler (taking 4 bytes) such as "PASU" for the UBC/Pascal compiler), and a 4 byte field that can be used for anything by the compiler (like a sequence number for type descriptions that require more than one record).

The MATCH2 field would contain type descriptions of parameters or elements of a common area in any format the compiler wishes to use. If a fixed number of parameters is required, the first thing to be included in the MATCH2 field could be the count of the number of parameters. The type-descriptions of each of the parameters would follow.

By definition, in FORTRAN one is allowed to "extend" COMMON areas. So, for example, the following is legal:

```
SUBROUTINE A ...
COMMON/BLAH/I,J,K,A
INTEGER I(10),J(4),K
REAL A
 ...
SUBROUTINE B
COMMON/BLAH/I,J,K,A,V,F
INTEGER I(10),J(4),K,F(3)
REAL A,V(3)
```

It is possible to properly check for legal FORTRAN COMMON with this model. Since the loader only compares the minimum of the lengths of the two records, the compiler can spit out TYP records associated with "BLAH" that will match, from subroutines A and B.

Since the number of parameters for "standard Pascal" procedures is fixed and the language is relatively strongly-typed, this model covers the type checking that needs to be done between two separately-compiled "standard Pascal" routines. I believe that the same thing goes for ALGOLW and ALGOL60. However, in PLUS, PL/I, FORTRAN, MDSI Pascal, and GCM there are a number of problems with it.

There are also special problems with ALGOL68 and ADA (why not look ahead). However, modular compilation of true ALGOL68 and ADA procedures is nearly impossible with the current loader, because of strict definitions and the ability to overload procedures in both languages. Maybe I shouldn't have mentioned them, because I'm certainly not going to talk about them.

a. In PLUS and GCM (and FORTRAN if you don't want to adhere to the standard), a procedure can have a varying number of parameters. If no "number of parameters" field is put in the MATCH2 field, a procedure can be called with fewer than the minimum number of parameters required by the procedure; but, if a "number of parameters" field is included in the record, the procedure cannot have a varying number of parameters.

In PLUS, one could require that the procedure declarations match exactly, but that would mean that a calling procedure would have to contain declarations for optional parameters for the called procedure whether they were used or not.

b. In FORTRAN and GOM, one can effectively slice an array by calling a routine with an array element not at the lower-bound. Also, the routine it is calling can slice the array by simply dimensioning it to be smaller than the passed array or array-slice. Not only is this possible to do, but I believe it is also legal as far as the FORTRAN definition goes. GOM, of course, is thoroughly ad hoc. I think the rule is: Anything you can do in the language is legal.

Therefore, with this model, the only practical thing that you can do with this is to simply check to see if the base types match on procedure calls.

c. In PLUS, a procedure can have a parameter with the type "pointer to unknown". Internally to the procedure proper, that parameter may be represented as a "pointer to <something>". This model can handle that situation adequately by having one TYP record produced per parameter. Then the procedure that had a parameter "pointer to unknown" would produce a short TYP record.

The same sort of solution applies to FORTRAN string constants, which can be coerced into any type over a procedure call. Like the "pointer to unknown" solution, this would require one record per parameter; but unlike the previous situation, the record corresponding to the parameter from the calling routine would simply not be produced. Thus there would be nothing for the called procedure's parameter record to match against, and the type-checking operation would succeed.

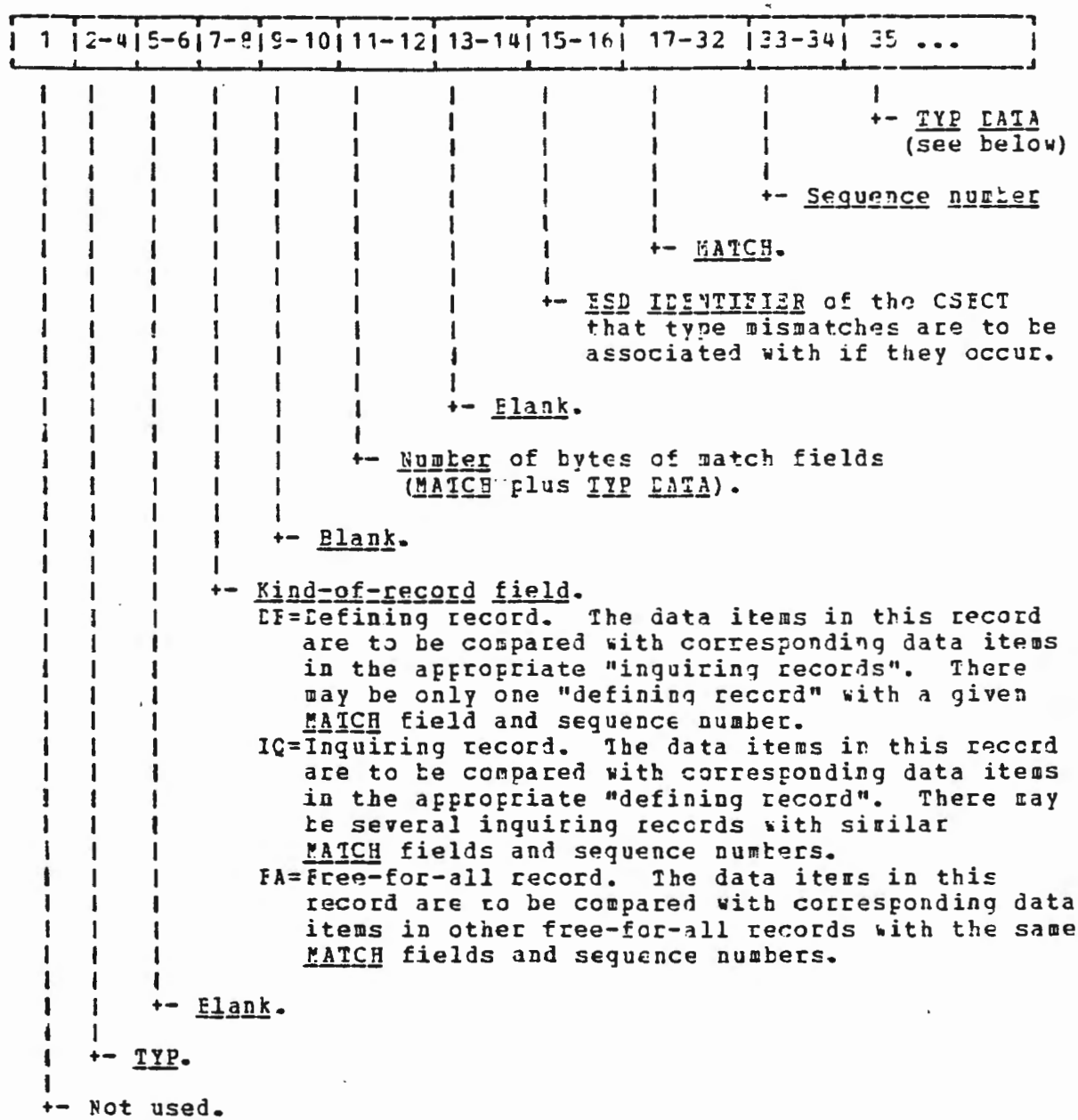
d. Actually, I lied about FORTRAN COMMON, before. Probably the most prevalent COMMON extensions are those like the example given above; however, an extension can also be of the following form:

```
SUBROUTINE A ...
COMMON/BLAH/I,J
INTEGER I(10),J(4)
 ...
SUBROUTINE B ...
COMMON/BLAH/I,J
INTEGER I(10),J(11)
 ...
```

TYP records of this model cannot practically handle this situation. We are thus left with a situation similar to that described in "b" above. Like the only reasonable solution for "b", we are limited to checking for base type compatibility only. So this model is becoming less and less helpful all the time. Restricting type-checking to base-type compatibility for COMMONs, will only determine error conditions in a few of the possible cases.

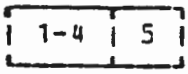
Proposed Format Number II

TYP Input Record



Sequence numbers: These start at 1 and are incremented by one for each subsequent record in the sequence. A sequence number of zero indicates the end of the sequence. This allows type definitions that are longer than one record. TYP data items may be physically split over more than one TYP record.

TYP Data Item



+- Flag field--(ERNNLCCC)

E=existance flag
0=this element should be considered missing. No other flag bits are significant.
1=this element exists. Other bits in this field are significant.

R=Required-to-exist field
0=if the TYP data item being compared to this one is missing, the comparison will succeed regardless of other bit settings.
1=if the TYP data item being compared to this one is missing the comparison will fail, regardless of other bit settings.

NN=Reserved for future expansion

L=Last-field-required field
0=If the data item being compared to this one is the last item in the sequence of data items on that record, the comparison succeeds, regardless of other bit settings.
1=Last data items being compared to this item are treated the same as normal data items.

CCC=Comparison field.
fields corresponding to this one on appropriate records will succeed if they are:
111=anything
000=comparison fails
100=greater than
010=less than
001=equal to
101=g.t. or equal
011=l.t. or equal
110=not equal to
this one.

+- Comparand field. This field is considered to be an unsigned fullword integer. Comparisons are made with other comparand fields that positionally correspond with this one and that fulfill the requirements of the "kind-of-record" field. Bits in the "flag" field of this data item define the use of this field.

TYP record processing algorithm:

```
if CPT TYPECHECK=ON
  then
  | if (SequenceNumber ≠ 0)
  |   then
  |   | if (SequenceNumber - 1  = LastSequenceNumber)
  |   |   then
  |   |   | print an error message;
  |   |   | LastSequenceNumber := SequenceNumber;
  |   |   | exit this mess (I don't have indent room)
  |   |   fi
  |   else
  |   | if SequenceNumber = 1
  |   |   then
  |   |   | CurrentKindOfRecord := KindOfRecord;
  |   |   else
  |   |   | if KindOfRecord  = CurrentKindOfRecord
  |   |   |   then
  |   |   |   | print error message;
  |   |   |   | exit
  |   |   |   fi
  |   |   fi
  |   fi
  | if LastSequenceNumber = 0 (i.e. this is the starting record)
  |   then
  |   | set up a node for this TYP description.
  |   | put this node at the beginning.
  |   else
  |   | append the contents of this record to the end of the
  |   |     node we are currently working with.
  |   fi
  | if SequenceNumber = 0
  |   then
  |   | scan our list of information (InfoList) for the info
  |   |     corresponding to the current MATCH field.
  |   |     Call it MatchInfo.
  |   | case KindOfRecord of
  |   |   DF:
  |   |   | case (KindOfRecord of MatchInfo) of
  |   |   |   DF:
  |   |   |   | if MatchInfo ≠ ThisRecord
  |   |   |   |   then
  |   |   |   |   | print error message
  |   |   |   |   | exit.
  |   |   |   |   fi
  |   |   |   FA:
  |   |   |   | print error message
  |   |   |   | exit
  |   |   |   IC:
  |   |   |   | perform comparisons of this record with MatchInfo
  |   |   |   |     see algorithm below.
  |   |   |   | Remove MatchInfo from InfoList
  |   |   |   | Insert the information on the current DF record
  |   |   |   |     onto InfoList
  |   |   |   esac
```

```
|   |   IC:
|   |   | case ("Kind" of MatchInfo) of
|   |   |   FA:
|   |   |   | print error message
|   |   |   | exit
|   |   |   DF:
|   |   |   | perform comparisons of this record with MatchInfo
|   |   |   |     see algorithm below.
|   |   |   IC:
|   |   |   | merge information on this item with information
|   |   |   |     on MatchInfo.  see algorithm below.
|   |   |   | Remove MatchInfo from InfoList
|   |   |   | Insert the merged information onto the InfoList.
|   |   |   esac
|   |   FA:
|   |   | if "Kind" of MatchInfo is IC or DF
|   |   |   then
|   |   |   | print error message
|   |   |   | exit
|   |   |   fi
|   |   | perform comparisons of this record with MatchInfo
|   |   |     see algorithm below.
|   |   | if this record is longer than MatchInfo
|   |   |   then
|   |   |   | Remove MatchInfo from InfoList
|   |   |   | Insert the current record onto the InfoList
|   |   |   fi
|   |   esac
|   fi
fi
```

Merging algorithm

```
for I
  from 1
  to
  | MAX( (LastDataItem of MatchInfo),
  |      (LastDataItem of CurrentRecord) )
  do
  | if (ExistenceField of MatchInfo(I)) = 0
  |   then
  |   | MergeInfo(I) := CurrentRecord(I) (with some twiddling)
  |   else
  |   | if (ExistenceField of CurrentRecord(I)) = 0
  |   |   then
  |   |   | MergeInfo(I) := MatchInfo(I)
  |   |   else
  |   |   | (LastFieldRequired of MergeInfo(I)) :=
  |   |   |      (LastFieldRequired of MatchInfo(I)) oredwith
  |   |   |      (LastFieldRequired of CurrentRecord(I));
  |   |   | Determine a lower and upper-bound for MergeInfo ...
  |   |   |      basically, this takes the ComparisonField of
  |   |   |      the two items and determines the range of values
  |   |   |      that would be acceptable on a comparison.  Note
  |   |   |      that it is possible to have the lower bound be
  |   |   |      greater than the upper-bound.  When this is the
  |   |   |      case, the only items that can compare successfully
  |   |   |      with this item are those that are last items
  |   |   |      (if LastFieldRequired = 0) or non-existent items
  |   |   |      (if RequiredToExist = 0).
  |   |   | if (UpperBound < LowerBound)
  |   |   |      and (LastFieldRequired = 1)
  |   |   |      and (RequiredToExist = 1)
  |   |   |   then
  |   |   |   | print error message (there is no way anything can
  |   |   |   |      match if this happens)
  |   |   |   | exit
  |   |   |   fi
  |   |   fi
  |   fi
  od
```

Comparison algorithm

```
for I
  from 1
  to
  | (LastDataItem of MatchInfo)
  do
  | if (CurrentRecord(I) doesn't compare with MatchInfo(I))
  |     or (MatchInfo(I) doesn't compare with CurrentRecord(I))
  |          (I am hoping that you can figure out how to do this from
  |          the format of the data items and the description of the
  |          merging algorithm)
  |   then
  |   | print error message.
  |   | exit.
  |   fi
  od
for I
  from
  | (LastDataItem of MatchInfo) + 1
  to
  | (LastDataItem of CurrentRecord)
  do
  | if ((ExistenceField of CurrentRecord(I)) = 1)
  |     and ((RequiredToExistField of CurrentRecord(I) = 1)
  |   then
  |   | print error message
  |   | exit
  |   fi
  od
```

I may be deceiving myself, but I believe that the problems inherent in the Format I model are resolved by the Format II model. A good way to find out is to simply take some sample constructs in various languages and show some TYP records that represent the type-restrictions for them. Here we go ...

```
SUBROUTINE ALPHA(I,J)
INTEGER I
REAL J(10)
```

MATCH	KIND	COMPARAND	E	R	L	CCC
ALPHA...FTN.0000	DF	1 (procedure)	1	1	1	=
		2 (# parameters)	1	1	1	=
ALPHA...FTN.0001	DF	1 (Integer)	1	1	1	=
ALPHA...FTN.0002	DF	2 (Real)	1	1	1	=
		10 (# of elements)	1	1	1	>=

Now lets look at something that will attempt to match this calling sequence:

```
INTEGER R
REAL Z(12)
CALL ALPHA(R,Z(3))
```

ALPHA...FTN.0000	IQ	1 (procedure)	1	1	1	=
		2 (# of pars)	1	1	1	=
ALPHA...FTN.0001	IQ	1 (Integer)	1	1	1	=
ALPHA...FTN.0002	IQ	2 (Real)	1	1	1	=
		9 (# of elements)	1	0	1	<=

Attempting to load these two programs together will fail, because the item 2's of the ALPHA...FTN.0002 records do not compare appropriately. Note, that if Z(2) were specified in the calling program instead of Z(3), the loading would succeed (as it should) because the two comparands would compare favorably. If, instead, the REAL J had not been dimensioned in routine ALPHA, the loading would again succeed, because the last ALPHA...FTN.0002 (IQ) data item specifies that the corresponding DF data item does not have to exist.

Now, lets try another hard problem (FORTRAN is such a pain sometimes). This time, we will attempt to have COMMONs match that have last-elements with different lengths. Here goes ...

```
COMMON/BLAH/I,J,K
REAL J,K(5)
INTEGER I(3)
```

MATCH	KIND	COMPARAND	E	R	L	CCC
BLAH....FTN.0000	FA	2 (Common)	1	1	1	=
		1 (Integer)	1	1	1	=
		0 (array)	1	0	1	=
		3 (# of elements)	1	0	0	=
		1 (Real)	1	0	1	=
		1 (Real)	1	0	1	=
		0 (array)	1	0	1	=
		5 (# of elements)	1	0	0	>=

```
COMMON/BLAH/A,B,C
INTEGER A(3)
REAL B,C(10)
```

BLAH....FTN.0000	FA	2 (common)	1	1	1	=
		1 (Integer)	1	1	1	=
		0 (array)	1	0	1	=
		3 (# of elements)	1	0	0	=
		2 (Real)	1	0	1	=
		2 (Real)	1	0	1	=
		0 (array)	1	0	1	=
		10 (# of elements)	1	0	0	>=

Alas, the problem that we once had with COMMONs (see "d" in the section on Format I type checking) is now resolved. The loading of the two routines containing these constructs will succeed, as it should.

1171 - 1173
1159 - 1163
1271 - 1284

Actually, I have yet to find a place in the "real world" that requires the use of the "this data item exists"; however, I can think of some future possibilities. Picture a language where (unlike PLUS and Pascal) structured types are compatible if all subfields are compatible in type (not necessarily that the structure has the same layout) and "pointer to unknown" types are always compatible with any other pointers. If you want an exercise to try your hand at, this is an interesting one. I'm not including it because I don't want to bore you (or myself) any more than I have to. If you want to know how it would be done, I'll be happy to talk about the solution.

Even though the "data item exists" field is not required anywhere, it can be used to avoid spitting out additional TYP records (and some of the time, this saves disk space).

General discussion

I believe that the second model discussed in this paper has the ability to perform sophisticated type-checking for all situations resolvable at load-time. If you find any holes in it, I would appreciate it if you would let me know.

It is obvious how this construct is implemented in the loader, but it would also be nice to have *OBJUTIL be able to handle them in a sophisticated manner. One idea that immediately drops out of this discussion is having *OBJUTIL optionally read TYP records and place them in the negative line range instead of the front. During this operation it could check to see that the TYP records match appropriately. This avoids the additional cost of having the loader process TYP records every time an object file is loaded.

In addition, it could place all "DF" TYP records in the positive line range of the object file, so they would be operational if this object file were later concatenated with another. "IQ" TYP records that were not yet resolved by "DF" records, and "FA" TYP records can be compacted into the minimum representation required (which is a maximum of two TYP sequences for a set of matching "IQ" records and one TYP sequence for a set of matching "FA" records) and placed in the positive line range, too.

The question that I would be interested in resolving is: Are other people interested in having the loader process records of this nature? If they are, lets do it! It has immediate applications to reducing headaches associated with even that un-esoteric language, FORTRAN. Since FORTRAN has extremely heavy use at all MTS installations and since a number of strongly-typed languages are beginning to make their way into "real-world" usage, it would seem that this concept could fill a very empty gap.