PRESERVATION OF CODE QUALITY IN SOURCE-TO-SOURCE TRANSLATION: BLISS -> C

by

Roy Styan

B.Sc., Simon Fraser University, 1979

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computing Science

SIMON FRASER UNIVERSITY
November 1985

APPROVAL

Name: Roy Hamilton Styan

Degree: Master of Science

Title of Thesis: Preservation of Code Quality in Source-to-Source Translation: Bliss -> C

Examining Committee:

Chairperson: Joe Peters

Lou Hafer
Senior Supervisor

Rob Cameron

Jay Weinkam

Robert Ito
External Examiner
Professor
Department of Electrical Engineering
University of British Columbia

Date Approved: 13 December 1984

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Preservation of Code Quality in Source-to-Source Translation: Blis-to-Blis → C

Author:

(signature)

Roy Styan

(name)

Jan 14/86

(date)

Abstract

A major problem created by diversity in computing tools is the portability of software. Programs written on one machine cannot be used on others unless a compiler for the original source is present. Many solutions to the problem have been proposed, but none have been universally accepted. One solution, which allows programs to be maintained in their new environment, is source-to-source translation.

In this thesis the concept of quality translation, in the first phase of a Bliss-to-C translator, is explored. A quality translation is one which produces comprehensible, as well as semantically correct code. For a Bliss-to-C system, preservation of quality must first come through the translation of data structures. Bliss is an untyped language with access algorithms for defining structures, while C is a typed language using data aggregates. Heuristics have been developed for both deriving type form context, and building data aggregates from the definition and usage of access algorithms. A scheme is presented for transforming the original source to a more abstract form, resulting in a few abstract functions which can be translated by hand, and an improved source which is easier to translate by machine.

Table of Contents

List of Figures

1. Portability Through Source-to-Source Translation

In an ideal computing environment a programmer has the freedom to exploit the natural parameters of a problem. He can choose the algorithm, the language and the machine to implement a solution. When starting from scratch this freedom is ideal. In theory, a task will be completed with the most up to date technology available. In practice, however, most jobs cannot be started from scratch and will be tied to an existing machine, with a choice of two or three languages. Implementations will rely heavily on existing software, be it the operating system or program libraries. Although some languages are fairly widespread (*e.g.*, Fortran), many are company, machine or installation dependent. As well as being non-portable, programs may be restricted to communicating with other software written in the same language. This limits the use of large program libraries to aid in the solution of a problem. As better machines and languages are created programs become dated, requiring substantial effort in maintaining their usefulness. In short, the available computing environment often restricts programming choices to a very small domain. To change the domain

> "it may be easier to rewrite the software system for the new machine or the new operating system rather than try to modify the old one."[1]

Software portability is not a new problem. Several solutions exist [Griffiths 79], including source-to-source translation. However, because of the degree of difficulty in producing a translator the idea has not been well explored. Attempts at such translations include Algol --> PL/I [Griffiths 79], Fortran --> Pascal [Freak 81], Pascal --> Ada and Ada --> Pascal [Albrecht 80]. All of the above algorithms have been independent pioneering attempts. Each translator presents its own set of problems, having relatively little overlap with another translation. Hence, with the exception of Albrecht [Albrecht 80], little work has been done on formalizing source-to-source translation. Indeed, if the only concern of a programmer is to get working code then the style of translation is somewhat irrelevant. In such cases each translator does have a unique set of problems, namely forcing rules in one language into a form acceptable to the other language. Preservation of semantic content is the only goal. However, translation is not a unique process and an essentially infinite number of semanticly correct programs can result from a given source. Several factors contribute to such diversity, including readability, efficiency (in both space and time), and maintainability. While correctness is a prerequisite to any translation, it should not be considered the only goal.

---

[1] [Peck 78], pp. 143

1.1. Quality Translation

The necessity for consideration of program quality during the translation can be demon- strated in two ways. First of all, large software projects are rarely static. When first written they perform a task with minimum variation from a straight and narrow path. As time goes on the code must be modified for improved performance, or to encompass a greater scope of the problem. Such reworking of code is the art of program maintenance. In order to maintain a program one must be able to understand it. Virtually every course in undergraduate computing science is devoted to training people to write comprehensible code. Hundreds of books and papers have been devoted to the subject. Such concepts as structured programming, top down programming, and data abstraction have all been introduced to try and get people to write readable programs. The art of computer programming is an art of communication! Yet this issue has been completely ignored by most translators. It is easy to do. There are enough problems just grinding code into the target language. However, if the issue of code quality is ignored the translation may well be in vain. If one cannot see what is going on in the target language the maintenance is impossible. Thus the program will have to be maintained in the original language. If this is the case, then what is the point of the translation?

Secondly, it is inevitable that a mindless translation will introduce gross inefficiencies into the code. Either the execution time of a program increases, due to auxiliary statements, or the program size increases. The latter is more likely to happen to languages with radically different data structures. Consider, for example, the data structure shown in figure 1. In an undirected translation between Bliss and C a single structure was found which could take the place of the fifteen smaller, more modular structures shown in appendix 1. The resultant code would still be correct, as the variable declarations would allocate structures with appropriate fields. Type restrictions would be satisfied, if necessary, as all pointer variables are of the same type. However, for any given variable the majority of space allocated to it would be wasted as only a few of the fields would be utilized. Also the new code would loose much of its legibility since the object represented by some variable could be one of fifteen. Clues identifying which object the variable stands for are spread throughout the code. In a quality translation structure "*node*" would have to be broken into smaller structures, one structure for each set of logically related fields, for clarity and efficiency.

Node:

fwrd (int 0)					
ounxt (int 0)	snnxt (node 1)	dfsons (node 1)	nlnxt (node 0) (node 1) (node 2)	dvnxt (node 1)	llval (node 1)
opinlst (node 1)	dfcros (node 1)		doC (node 1)	dohead (int 0)	llnxt (node 1)
ousrc (node 1)			dvsval ()	doL (node 1)	
snsrc (node 1)	dfidnum (int 0)	dvcode ()	dotail (int 0)	dofhead ()	sheqtedto (node 1)	ouprnt (node 1)
snstype ()	dfVTptr (node 1)	dofheadadr (node 1)	dotailadr (node 1)	dohdlst (node 1)	shLptr (node 1)	doGCA (int 0)
opdfptr (node 1)					
oudfptr (node 1)					
dvtype ()		nlprv (int 0)		dfdad (node 1)	docedge (node 1)
dveqtedto (node 1)	snvarlst (node 1)	dffwrd (node 1)	doheadadr (node 1)	doF (node 1)	shdead (int 0)
doLrev (node 1)					
vtoutlst (node 1)					
ouvarlst (node 1)					

Figure 1: Example of undirected data structure translation

Program quality is not something that lends itself easily to definition. The problem is its subjective nature - quality depends on an individual's style. However, Peck and Schrack [Peck 78] have outlined several criteria for quality software. These are:

* correctness,
* reliability,
* robustness,
* readability and documentation for users and maintenance,
* small, easily understood modules,
* the ability to communicate with man,the operating system and other software,
* efficiency with respect to time and storage, and
* portability.

In terms of translation some of the points are irrelevant, as either the original code fits the criteria or it doesn't. On the other hand, the translation will affect the program's readability, efficiency, portability, and its communication ability. These factors should also be considered goals when designing a source-to-source translator.

Freak [Freak 81] has done just this in his Fortran --> Pascal translator. As well as preserving semantic content he forces structure onto a Fortran program. His idea of a structured program is simply a "one entry, one exit" philosophy. Each block of code must have a single entry point and a single exit point. Although this philosophy can be followed in Fortran programming, the dependence on goto's to alter sequence of control flow makes it hard. Using a "Boolean flag technique", Freak eliminates goto's from the Fortran program, replacing them with Pascal while loops. Translation has now gone beyond simple semantic analysis. His programs must deal with the problem of code rearrangement, perhaps introducing auxiliary variables or even auxiliary statements to achieve correct translation. However, his goal is not only correctness, but provability. The end result is a program of higher quality than the original source.

1.2. Quality Control Through Data Structure Translation

Several methods and tools exist which try to standardize the production of quality code [Jackson 75, Bergland 81, Linden 76]. Most of these methods are not useful to translators, however, since they consist of techniques for dealing with code before it is written. Such methods as functional decomposition, data flow design, and programming calculus [Bergland 81] end when the program

has been implemented in the original language.

A fourth method, data structure design [Bergland 81], proves itself to be useful for program maintenance as well as design. The method requires one to define structures to represent the data and then define the program as set of operations to perform on the structures. Both Jackson and Linden claim changes to a program designed in this fashion will be simple and localized in nature. It is because languages are essentially an abstraction of a machine [Rowe 81] that the method works. There are control abstractions, representing program sequences, and data abstractions, representing the external environment and the operations which can be performed on the representations. Control sequencing is fairly standard among procedural languages. Although the syntactic form may vary the essential nature of control is the same.

On the other hand, the method of implementing data abstraction is the flavor of a language. Each has its own way of building data aggregates, and its own set of operations to be performed on them. The higher the level of the language the greater the abstraction of data and suppression of implementation detail. The concept of data abstraction goes beyond the basic data types and aggregates available in a particular language, however. Programs are now written with abstract data types as units of modularity. For example, a program segment will reference a piece of data through a subroutine call. Thus the particular implementation of the data is unknown to the program. This technique greatly simplifies program maintenance. In order to change a representation one simply rewrites the implementation of the abstraction modules which manipulate the data structure, leaving the interface with the calling routines untouched.

1.3. A Data Structure Approach to Translation.

The notion of abstraction reflects the fact that a program is simply a machine for transforming data from one form to another. Data is grouped into structures which become abstract representations of the objects being manipulated, and control structures become the operations which produce the desired transformation. Thus objects and operations go hand in hand, and a well written program makes a clear distinction between the two. If, under transformation, the data structures are fused or fragmented, then their meaning as objects is lost and the operations performed on them no longer make sense. Under these conditions a program's quality is greatly reduced. It is the latter concept that leads to the central hypothesis of this work, namely that a quality translation can only be achieved if the data structures are correctly translated. In other words quality code cannot be achieved unless a natural representation for the data can be found in the target language. Several levels of knowledge would be needed to deduce this natural

representation. A system would have to know the language's semantics in order to deduce the program's primitive data structures, such as data aggregates. For untyped languages this may not be a simple task. Also, an understanding of the pragmatics, in combination with semantics is necessary to build an understanding of the model the programmer is working with. Pragmatic understanding of the source language would produce the abstract model, while that of the target language could express the model in terms of its own data structures. This model is shown in figure 2.

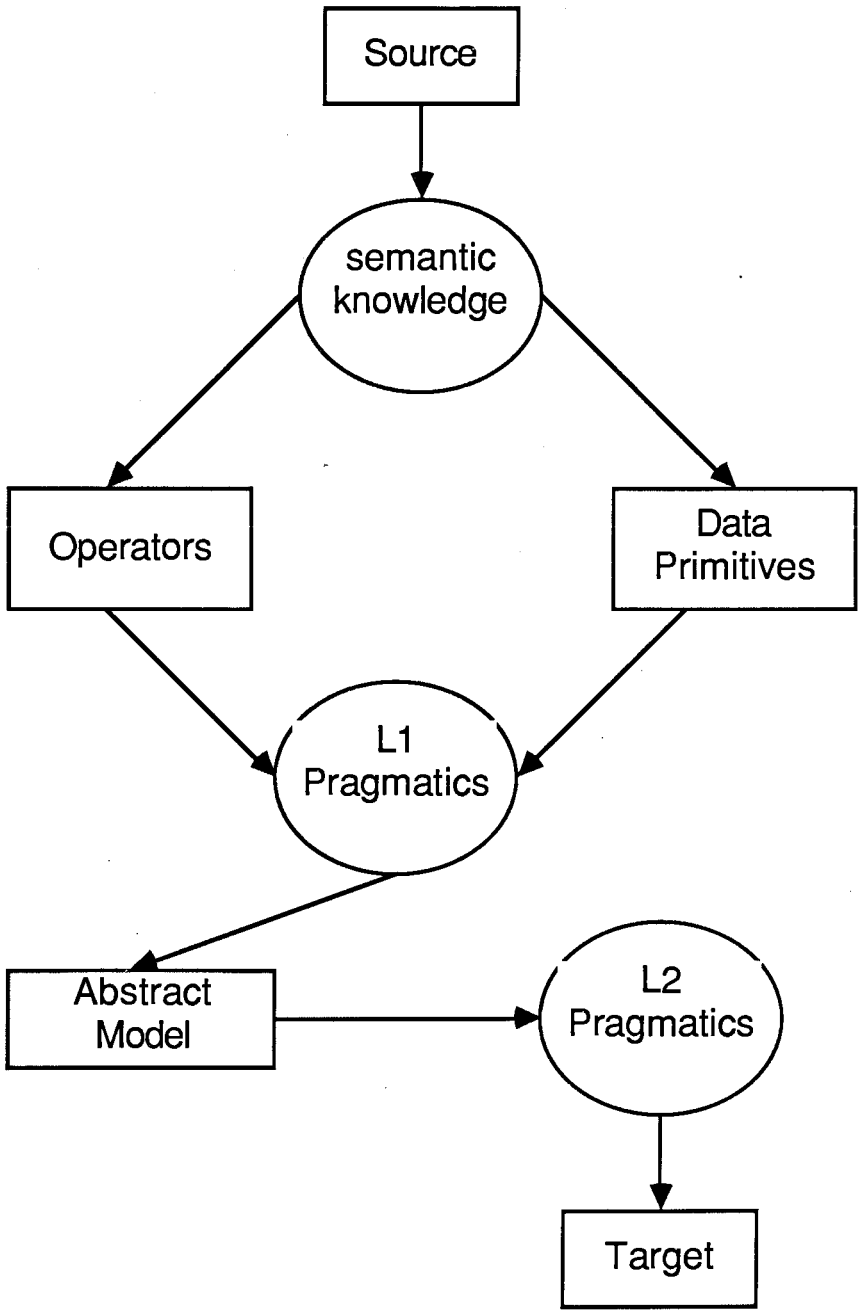


Figure 2: Translation System

Whether this model of translation can be fully automated is not clear. The task of building this knowledge into a translator is quite formidable, if not impossible. One possibility for coping with the complexity, though, would be to try and transform certain segments of a program, possibly identified by hand, to more abstract forms. The abstract modules could later be implemented in the target language by hand, if necessary, as shown in figure 3. Such a highly interactive system could substantially ease the burden of the translator.

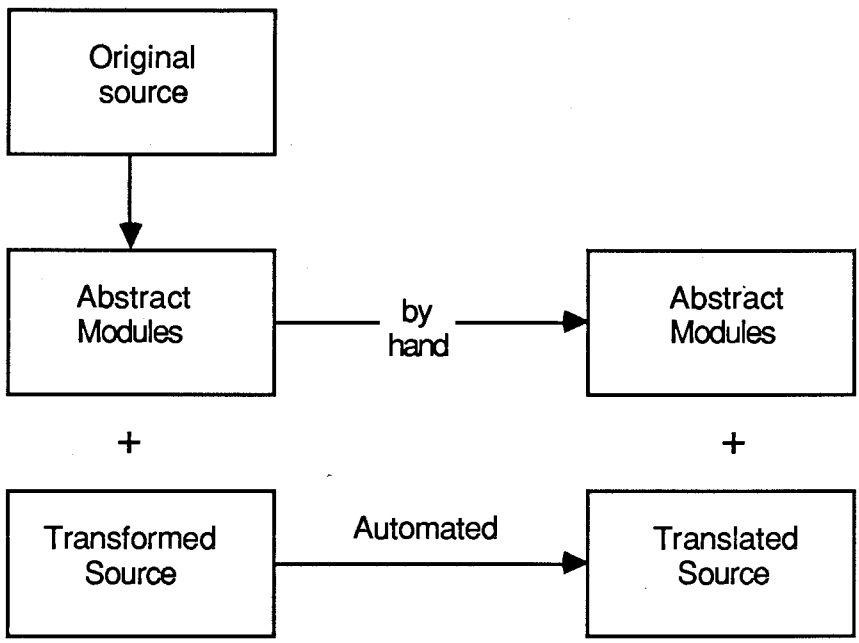


Figure 3: Alternative Translation Model

The first problem in the translation system, though, is to deduce the data primitives in the source language. As mentioned above, for unstructured, untyped languages this is a non-trivial task. The remander of the thesis will be concerned with this point.

1.4. Quality Measurement

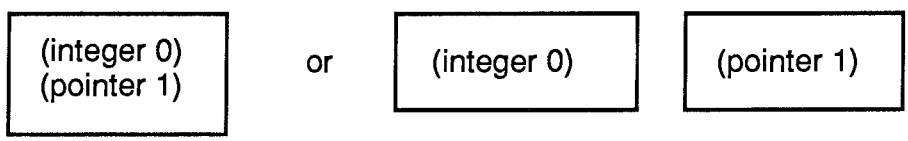
Software quality is subjective, rather than objective in nature. Anything subjective in nature is difficult to specify in an algorithmic manner, a requirement for automation. Thus, if a translation system is going to be based on software quality, then a method for quantitative evaluation of quality must be developed. The need for software metrics has long been recognized [Halstead 77, Harrison 82, Brown 81]. However, the science is still young and a good set of metrics which define precise, standard, well understood code have not been developed.

Perhaps the best known metrics are those developed by Halstead [Halstead 77] in his thesis on software science. Such things as program length, number of operators, and number of operands are counted and used to calculate certain quantitative measures. Halstead's contention is that these measures form the basis for comparing the quality of various programs. If code quality is to be maintained under translation then these measures should remain invariable.

Halstead's ideas on program metrics are only useful for control structures. Concepts such as operands, operators, and program length are usually thought of in terms of executable code. Data structures have neither operands or operators, nor does their length necessarily reflect good or bad code. It is necessary therefore, to decide on some other metric which somehow indicates the quality of a program's data structures. For example consider the data structure in figure 1. Keeping in mind Peck's [Peck 78] quality criteria it is clear that this structure fails to satisfy the goals of:

* readability,
* small, easily understood modules, and
* ability to communicate with man.

The failure here is because many untyped structures have been forced into a single typed one. This results in an extensive number of types for a single field, requiring many distinct field names to identify them. In essence, the structure represents several different structures and trying to separate them would give even the most seasoned hacker ulcers. If this structure were somehow separated into its component parts, each part having uniquely typed fields, then much of the confusion could be avoided. For example, consider a simple Bliss access algorithm which retrieves a single field. Suppose several variables have been associated with the algorithm, some to be used as integers, others as pointers. In order to satisfy Peck's criterion, a programmer should use two mnemonic macro names to represent the distinct types. Program readability will be satisfied as well as the Bliss syntax allows. The declaration itself gives no hint as to the variable's use, though. If this algorithm were to be translated into a typed language one of two possibilities would result. The access algorithm would translate into either a single structure or two structures, one with an integer field, the other with a pointer field.



In the former case a type union would be necessary and a person reading the program would

wonder why the union was chosen for this particular structure. If two structures resulted from the translation no unions would be necessary and it would be obvious if a variable were an integer or a pointer. Peck's criterion would be more than preserved in this case. It should be possible, then, to use the number of type unions within a data structure as a metric. The more unions added by a translation, the poorer the quality of the resultant code.

1.5. Issues in Code Translation

For any particular translation system the issues can be classified as general, language specific, or machine specific. For instance, all translators must have some purpose, or ultimate goal, to direct the translation. The nature of the algorithms and heuristics used will be affected by the desired outcome. Thus if the purpose is simply to translate code at all costs then the quality issues discussed above can be ignored, and the translation will proceed strictly on a syntactic and semantic basis. However, as the goals become more complex the system's knowledge of quality will increase. Quality issues are not language dependent, and should remain constant from language to language.

Given that the system has some specific criterion by which to judge a successful translation the second issue, namely an understanding of semantics, comes into play. A successful translator should be able to extract an understanding of the data structures from the code, as well as the operations performed on these structures. The more abstract this understanding, the more successful the system will be at doing its job. Of course, to abstract this knowledge, a translator must have an intimate understanding of the languages involved.

In some languages, such as pascal and C, the data structures are explicitly laid out, and the translation system can easily extract them. Others, such as Bliss, do not explicitly state the form of their structures, rather the whole picture is spread throughout the code. Nor does Bliss enforce that programmers remain consistent with their structures. A block of data is simply a series of bytes to manipulate as one sees fit. At this level it becomes very difficult for a program to abstract, hence a fully automated system is harder to ensure.

If a translator cannot gain an abstract notion of what the program is trying to do then it must rely on semantic and syntactic knowledge, and the relationships between the various syntactic constructs. For languages which are structurally similar, these relationships may be nothing more than one-to-one mappings and the translation can proceed without any deeper understanding of the code. When languages differ radically though, abstraction is a necessity.

Finally, the translator must have some knowledge about the two machine architectures involved. In languages where data primitives include bits and bytes, architectural dependency will become a big issue. However, even languages with a lesser dependency can be affected by a move to a different machine. For example, the size of an integer and floating point number in Bliss is 36 bits, a reflection of the fact that the language was designed for the DEC-10. The equivalent primitives in C, on a Vax 750, are 32 bits. Obviously any programs that rely on the extreme values allowed by 36 bit architectures will be thwarted by the smaller machine. Similarly, in languages that allow bit fiddling, the larger architecture may create data structures, such as bit vectors, that have no corresponding structure on the other.

Each of the issues presented above create some challenging problems for an automated translator, and it would be difficult to deal with all three in any one thesis. However, they are somewhat independent, and certain problems can be dealt with while ignoring others. In the following chapters the problems encountered in a Bliss-to-C translator are reported. Chapter two discusses the intended scope of the thesis, and proposes some assumptions on which to base the translation. Also, an in depth comparison of the syntax of Bliss and C is made, exposing the aspects that must be inferred indirectly from the Bliss source. Chapter three presents some algorithms to cope with data structure translation. Finally, in chapter four the algorithms are applied to the data in appendix 1, and the results discussed. Conclusions and future prospects are presented in chapter 5.

2. A Bliss-to-C Translator

As an illustration of the concepts discussed above the first phase of a Bliss-to-C translator was written. Both Bliss and C are "implementation" languages; that is, they are designed as high level languages suitable for writing system software. So much for their similarities. Bliss was written with a specific machine in mind, the DEC-10, and deliberately includes constructs closely tied to the DEC-10 processor architecture. Although this makes Bliss a very efficient language, portability becomes a major problem. On the other hand, C is becoming a very popular language and as such can be found on a wide variety of machines. Code can be ported from one system to another with relatively little effort. It is not unreasonable then, to spend a little time on an automated translation between the two languages. With valuable production software lack of portability can be costly.

A second, more fundamental reason for a Bliss-to-C translation is their lack of similarity, which poses interesting translation problems. For instance, a Pascal-to-C translation would be relatively straight-forward, both languages having very similar data structure mechanisms. With the radically different mechanisms of Bliss and C all kinds of problems arise when trying to produce quality code.

A full scale translator would require a major undertaking, far beyond the scope of this thesis. However, the problem breaks neatly into two parts, translation of data structures and translation of control structures. Both parts have problems requiring non-trivial solutions, but the translation of control structures is in part dependent on the translation of data structures. Thus the discussions that follow are concerned exclusively with the conversion of Bliss data structures to C data structures.

2.1. Goals

Earlier, the importance of having goals to direct the search for a good translation was discussed. Goals can be a set of heuristics, or metrics, by which to judge the quality of the output. These should be chosen carefully, as they will form the basis on which the target code is built.

In the Bliss-to-C system it was important to have a metric which measured the success of translating a program's data structures. It was felt that a data structure directed translation would lead to quality code for two important reasons. First, because the data representation forms the heart of any program, and secondly because of the intrinsically different ways Bliss and C handle data representation. Bliss uses "access algorithms" to access particular data fields, and is

completely untyped. C structures, on the other hand, are aggregates of named fields, each field of a specific type. Data in C is accessed by specifying the name of the desired field. With such different representations it is easy for a translator to stray from a good solution without direction in this area, and it is important to include a metric which measures the program's ability to deduce a set of C-like aggregates from Bliss code. For a full blown translation system several metrics would be needed to measure its success. Each would measure some well defined component of the overall program's implementation. For instance, one metric could measure the Bliss program's deviation from some accepted standard, while another could measure the target code's deviation. As the similarity of these two metrics would indicate, it would be difficult to distinguish, in the end, how well a translation succeded. A badly coded Bliss program would yield a badly coded C program. Mistakes introduced by the translator would be hard to detect. For this reason two simplifying assumptions were made. First, it was assumed the Bliss programmer chose to define a set of data objects and that each variable in the program represented a single object. Secondly, it was assumed that the type of each variable or field did not change throughout the Bliss program. These assumptions are not unreasonable. Programmers need to structure their programs in such a way that the code is easy to follow and the assumptions are really only rules of good programming practice.

With these assumptions a "union count" metric can be used to direct the translation. The count is simply the average number of variables with union types for a given Bliss module. A large union count implies a large number of unions have been introduced by the translator. Minimizing this count tends to merge structures together, so that ultimately all structures become one (as in figure 1) and all pointers have the same type. Thus the metric must be combined with another which offsets the merging. A simple number to do this is the inverse number of structures found, yielding a metric that is the sum of the union count and the inverse count. The smaller the value, the better the translation.

2.2. Syntax

C is a typed language, and as such has a richer set of declarations than Bliss. These are "used to specify the interpretation C gives to each identifier" [Kernighan 78]. Unlike Bliss, C variables must have an explicit declaration that requires a lot of information from the programmer. A translator must somehow accumulate from the Bliss code the attributes that are normally programmer supplied. To find out which attributes are needed it is necessary to compare the syntax of a C <declaration> to that of a Bliss <allocation declaration>. Figure 4 shows the necessary C syntax, while figure 5 shows the Bliss equivalent.

Since Bliss is a block-structured language the <allocation declaration> simply introduces a variable to the current scope. Each name is bound to a storage segment of a fixed number of words, each word consisting of 36 bits. Segments are used to contain data, generally integers, floating point numbers or pointers to other data. However, to a Bliss compiler the word merely contains a pattern of bits. No <type specifier> is required.

```
<declaration> ::=
        <decl specifier list>
        [<declarator list>]

<decl specifier list> ::=
        <decl specifier>
        { <decl specifier> }

<decl specifier> ::=
          <type specifier>
        | <sc specifier>

<sc specifier> ::=
          <auto>
        | <static>
        | <extern>
        | <register>
        | <typedef>

<type specifier> ::=
          <char>
        | <short>
        | <int>
        | <long>
        | <unsigned>
        | <float>
        | <double>
        | <structure union specifier>
        | <typedef name>
```

Figure 4: C declarations

What do the two declaration specifications have in common? Ignoring the <register allocation>[1], which has no C counter-part, an <allocation declaration> consists of an <allocate type> specification and a list of segment names. The C declaration consists of <decl specifier list> and a list of declarators (identifiers). It turns out that there is a one-to-one correspondence between Bliss's <allocate type> and C's <sc specifier>. Both are used to described the type of allocation desired: auto/local declarations cause allocation and deallocation of storage upon block entry and exit; static/own declarations allocate storage for the block throughout program execution; external/extern declarations are messages to the compiler that storage has been allocated elsewhere; register declarations indicate the storage is in one of the machine registers. C variables declared outside of a function definition are by default global, unless preceded by a <static> declaration. In the latter case the variables are hidden to other files.

<allocation declaration> ::=
    <variable allocation>
  | <register allocation>

<variable allocation> ::=
    <allocate type> <segment_names:msid list>

<allocate type> ::=
    <own>
  | <local>
  | <global>
  | <external>
  | <register>

<msid list> ::=
    <msid element> {, <msid element>}

<msid element> ::=
    [<structure:name>] <size chunks>

Figure 5: Bliss declarations

[1] The <register allocation> is one of Bliss's machine-dependent features. It allows for the specification of particular processor registers for efficient handling of global variables. As C is designed as a portable language no such feature exists.

C's <type specifier> is virtually missing from the Bliss specification with one exception, the optional <structure:name> in the <msid element>. In a Bliss program the <structure:name> refers to the name of an access algorithm defined earlier in the program. The closest C counterpart is the <typedef name> in the <type specifier>. These two "structure" mechanisms will be discussed later in section 2.3.

C's <declarator list> and Bliss's <msid list> are the parts of the declaration which name the allocated segments. The syntaxes are shown in figures 6 and 7 respectively.

```
<declarator list> ::=
        <init declarator>  {, <init declarator>}

<init declarator> ::=
        <declarator> <initializer>

<declarator> ::=
          <identifier>
        | <bracketed declarator>
        | <pointer declarator>
        | <function declarator>
        | <array declarator>

<bracketed declarator> ::=
        ( <declarator> )

<pointer declarator> ::=
        * <declarator>

<function declarator> ::=
        <declarator> ()

<array declarator> ::=
        <declarator> ' [ ' <constant expression> ' ] '
```

Figure 6: C declarator list

Again the Bliss syntax is simpler, and requires less information than the C counterpart. The <msid list> is essentially used to name segments and declare their size (which defaults to one word)

```
<msid element> ::=
        [<structure:name>] <size chunks>
<size chunks>  ::=
        <size chunk> {: <size chunk>}
<size chunk> ::=
        <idchunk> [ '['<expression list>']' ]
<idchunk>  ::=
        <name> {: <name> }
```

Figure 7: Bliss msid list

A C <declarator list> is used to name segments, initialize them, and provide further information to the compiler about their use. The latter is something the Bliss compiler doesn't require. For instance, a <pointer declarator> tells us the variable is to be used as a pointer to a specific object (the object given by the <type specifier>). A <function declarator> gives the type of the object returned by the function, and an array declarator tells us the type of each element in the array, as well as the size of the array. Notice that the only size information required by the C compiler involves arrays. All other structures are well defined groupings of primitive data types, hence the size can easily be calculated by the compiler.

Essentially the difference between Bliss and C declarations is the lack of "type specifications" and "use information" in Bliss. For most variables "use information" boils down to whether or not the variable is a pointer, and finding its level of indirection. This is the information that must somehow be derived before translation can take place, and is what will be referred to as type in the following discussion.

2.3. Data Structures

Unlike most block structured languages, Bliss has no implicit data structures. Instead, the language provides a method for defining an explicit data access algorithm. The algorithm accepts a base address, one or more parameters, and produces the address of the desired field. For example, consider the following specification:

```
STRUCTURE list [p,s] = [1] (.list)<.p, .s>;
```

The keyword STRUCTURE indicates that the following expression is an access algorithm named list, with two input parameters p and s. Following the equal sign is a bracketed expression (the value 1) which indicates the size of a variable declared as "list" will be one word. Finally the actual access algorithm, in this case a simple pointer expression, is defined. The algorithm returns the address of a bit field of size s, starting at position p in the accessed word. In order to invoke the mechanism, a programmer would use a data access expression such as:

```
.x[18,18]
```

If x has been declared as 'list', this expression returns a bit field of size 18 starting at bit position 18 within the word with address named x. (Note, in Bliss the "contents of" operator is a "."). Thus the expression .x would read "the contents of the address named x"). What makes this mechanism powerful is the fact that the expression in the access algorithm can be as complex as the programmer desires.

With such a simple mechanism for defining data access operations one might be tempted to view Bliss as a good language for implementing abstract data types. For example, figure 8 shows two access algorithms for performing operations on lists.

```
MACRO next = 0,18$,
      prev = 18,18$;

STRUCTURE list [p,s] = [1] (.list)<.p,.s>;

STRUCTURE nth [n] = [1]
          (LOCAL list : temp;
           temp <- .nth;
           INCR i FROM 1 TO .n DO
              (temp <- .temp[next]; ))
```

Figure 8: Access algorithms for list operations

Algorithm "list" would be used, in conjunction with macros "next" and "prev", to access the linking fields of a list, while "nth" is an algorithm to yield the nth element of a list. Note that it is entirely up to the programmer to ensure that the links occupy the first word of a list element. In practice such "high level" data accessing mechanisms tend to be clumsy. In order to access the nth element of something previously declared as list, a new block must be opened to allow a declaration which associates the access algorithm "nth" with the variable. Normally a programmer would abandon the access algorithm approach in favor of a macro.

The concept of data abstraction requires three important components [Linden 76]. One must be able to define the representation of a type as well as operations to be performed on the type, and one must be able to protect the representation from unauthorized operations. Although Bliss has very powerful and flexible methods for defining representations and operations, it has no way of enforcing protection. Lists can be treated as trees, queues or even integers. Unauthorized usage may be a hacker's paradise, but it inevitably leads to great confusion when trying to decipher the meaning of a variable.

C's method for building data representations (called data aggregates) is a little more rigid than Bliss's. An aggregate is either a C structure or an array. Kernighan [Kernighan 78] defines a structure as "an object consisting of a sequence of named members". Each member can have any type, which is declared in the structure specifier. Syntax for the structure specifier is shown in figure 9.

<struct or union specifier> ::=
    <struct or union> [<identifier>]
    ['{' <struct decl list> '}']

Figure 9: C structure specifier

For example, in the following C declaration the typedef mechanism allows us to attach the name link to the structure, which has one field declared as a pointer to type node (an earlier "typedef") and the other declared as a pointer to itself.

```
typedef struct  linked
 {  node          *value;
    struct linked *next;
  } link;

link x;
```

If data aggregates are allowed, then operators to access elements of the aggregates must be available also. Unfortunately the flexibility of the access algorithms definable in Bliss is not attainable in C. Only four primitive access operators are available, and one must work within the bounds dictated by them or combinations thereof.

In terms of data abstraction C cannot admit to the power of Bliss's access operators, but as a typed language it can provide more protection. Only variables of the same type may be equated, primitive structure operations are restricted to structured variables, and use of primitives is restricted to their declared type. Of course in C one can always get around these restrictions by the use of casting, union of types, or by defining function operators. C does not check that an argument type matches the parameter type in a function call. It is only through these relaxations that there is any hope of translating Bliss into C.

As mentioned above, C has only four primitive access operations, namely:

1. the member operator ".";
2. the indirect member operator "->";
3. the array member operator "[<expression>]";
4. the dereference operator "*".

Each operator represents an implicit data access algorithm, and they may be combined in an arbitrary fashion to achieve an explicit data access. It is useful to define these operations in terms of access algorithms. The equivalent Bliss access algorithm is then easily identified. In general, the C member operator corresponds to the access algorithm:

$$\text{base} + \sum_{i=0}^{j-1} K_i \qquad (1)$$

Here the k_i's are constants representing the number of words the ith field occupies, and j is the field to be accessed. These constants are unimportant to a C programmer, as the compiler takes care of such details. In Bliss, however, the programmer is responsible for supplying the constants to the access algorithm. For example, a simple Bliss algorithm which achieves the member operation is:

```
node [w,p,s] = [w] (.node + .w)<.p,.s>;
```

The fields, specified by different values of w, p and s can be defined by macros such as *prev* and *next*. With the data access .x[next], where *next* is defined as "0,18,18", one would get an 18 bit value representing "the contents of the address with name x", starting at bit 18. (It should be pointed out that the half word (18 bits) is a special value in Bliss, the size of an address on the DEC-10). Of course, each field can represent a value of any type; the interpretation is not type restricted as in C.

The second C access mechanism, the indirect member operator, is used with pointer variables. It accomplishes the same thing as the member operator, with an extra level of indirection. For example, if variable x is declared "link *x" then it is interpreted as a pointer to structure link. Each member of the structure x points to is accessed with the indirect member operator, so x->next is an expression representing the "next" field.

The difference between the member and indirect member operators is a level of indirection. A simple change in equation (1) yields the access algorithm:

$$\text{indirect base} + \sum_{i=0}^{j-1} K_i \qquad (2)$$

Again, a simple Bliss algorithm for this operation would be:

```
node [w,p,s] = [1] (..node + .w)<.p,.s>;
```

An array member operator is one method for accessing elements of an array in C. The expressions between square brackets represent the indices of the desired element, hence an access such as *a[i]* yields the ith element of array *a*. C allows an alternate interpretation of an array which is more in line with the access algorithm concept. An array is simply a contiguous block of storage with each field representing a different element. If *a* contains the address of a block then the expression **(a + i)* represents the ith element. Thus an array is a special case of equation (1), each k_i having the same value, and j taking on the role of an index.

Finally, consider C's dereference operator. Suppose variable x has been declared as "int **x" (x contains a pointer to a pointer to an integer). The integer value represented by x can be accessed by the expression "**x". Like Bliss's "." the "*" is a "contents of" operator. In fact, the correspondence is one-to-one.

With only four access operators C's system for data access may seem primitive. It is the combination of these operators that lends power to building data structures. One can create nested structures, nodes linked in various fashions, arrays of structures, structures of arrays and so on. One task of the Bliss-to-C compiler is to interpret a Bliss access algorithm, and find the C equivalent. With an arbitrary access algorithm the best one could hope for would be a C function which mimics it. Of course this would lead to program inefficiency as well as reduce the overall readability of the code by introducing an excessive amount of type casting, since a function must return a specific type. It would be best if a valid C structure could be found to represent an access algorithm. Fortunately, most programmers use the types of aggregates mentioned above in their applications. Recognition of these types should suffice 99% of the time. Thus one should concentrate on Bliss algorithms that map into C aggregates. Any of the four Bliss structures discussed above (or minor variations thereof) should be easily recognized. But what about combinations?

To the uninitiated, the structure in figure 10 may appear a bit obscure (as the authors readily admit in [Wulf 71]) but in fact it is intended to be used with the data structure illustrated in figure 11.

```
STRUCTURE ITEM [i,j,p,s] =
 case .i of
  SET     (.ITEM)<.p,.s>
          (..ITEM + .j)<.p,.s>
          (...ITEM + .j)<.p,.s>
          (.(..ITEM + 1) + .j)<.p,.s>
  TES;
```

Figure 10: A more complex access algorithm

Here the access algorithm is actually a case statement, and the expression applied at invocation depends on input parameter i. However, each individual expression is equivalent to one of the C primitive access mechanisms. Case one is a special case of equation (1). It would be interpreted as a structure one word long, with bit fields given by p and s.

local ITEM x;

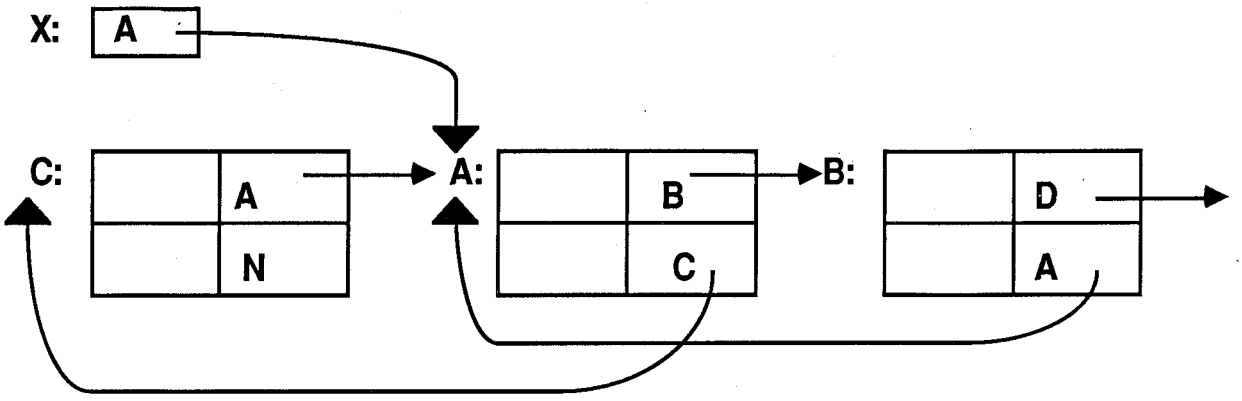


Figure 11: Structure described by figure 10

Cases two and three represent pointers to structures j words long (the range of j cannot be known until the entire program has been analyzed), each with an increasing level of dereference. Slightly more complex, case four can be interpreted as follows. The first expression (..item + 1) yields the second word in a structure pointed at by *item,* while the latter word is a pointer to a structure j words long. Without any other information, little else can be determined about the intended structure. However, a safe assumption is that the cases were meant to be used in conjunction with each other, rather than as single, isolated structures. Each case represents a different node in the same graph, the nodes furthest from the root being accessed by the expressions with the highest level of dereference. For example, the algorithm of figure 10 represents the directed graph of figure 12. In fact, figure 12 is simply a generalization of figure 11, certain links being absent. The complete specification of figure 11 is not required by the structure declaration.

In general, the case statements of an access algorithm can have arbitrary expressions. If, however, the expressions with a lower level of dereference can be shown to be subexpressions of those with a higher level of dereference then the access algorithm can be thought of as a method for accessing an arbitrary node in a directed graph. Expressions with i levels of dereference access nodes at distance i-1 from the root. For instance, to access a node at distance i (N_i) one must go through some N_{i-1}, which is accessed through expression j say. Then expression j must be a subexpression of k, the expression which access's N_i. In fact, this is the case for the expressions in figure 10. Case 1 is a subexpression of cases 2,3 and 4 (eg. case 2 can be regarded as .(case 1) + .j). Similarly case two is a subexpression of case three (set j=0 in case 2), and case four (set j=1 in case 2), which implies the branch in figure 12. Until the range of j is known, the field names

are found, and the field types are deduced (see section 4) further specification of structure *ITEM* cannot be known. It is only through context that there is any hope of deducing that the variable x is a pointer to a linked list.

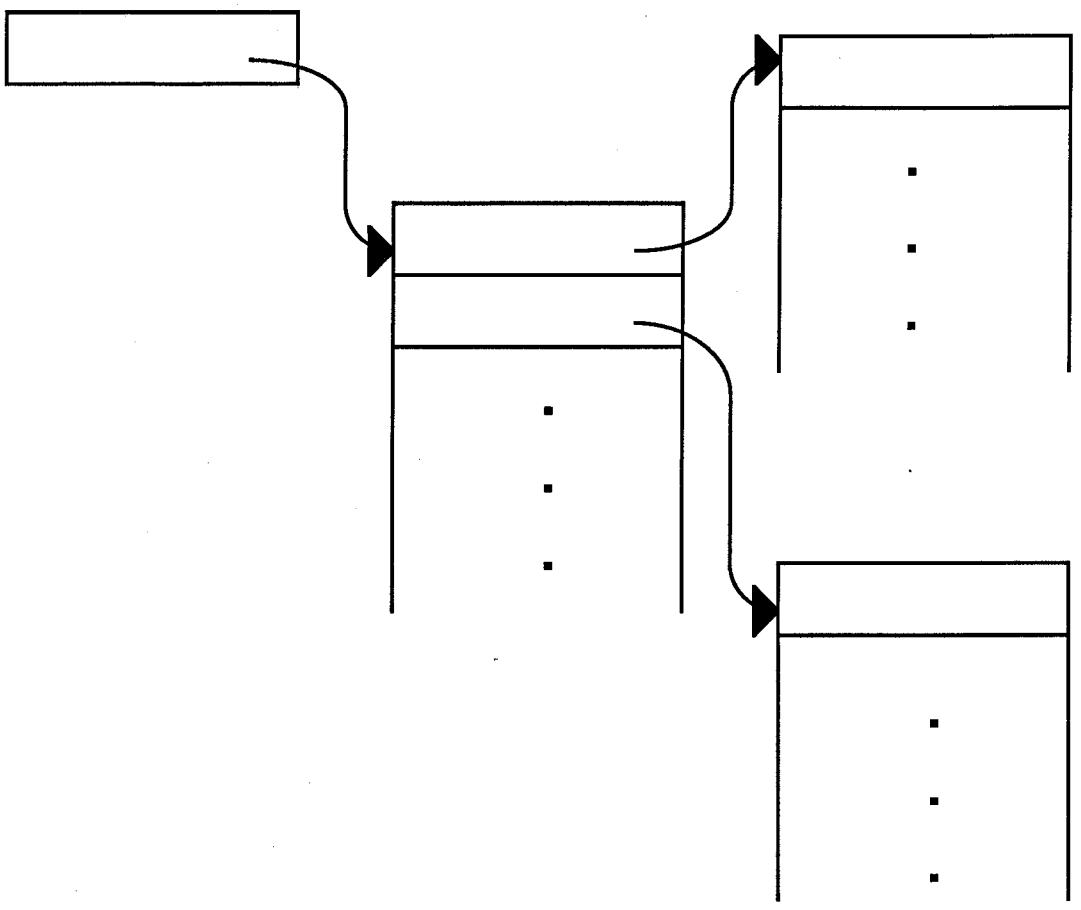


Figure 12: Structure represented by figure 10

3. Implementation

A translation of data structures can be considered completed on the successful building of a symbol table for the target language. Each identifier in the original source should have an entry in the table, along with a complete set of attributes to describe its use. In terms of a Bliss-to-C translation the symbol table must gather the information supplied by Bliss declarations, as well as the additional information required by the C compiler. Thus the first pass of the translator concerns itself with determining variable type and deriving C structures from Bliss access algorithms.

3.1. Determining Type

The C compiler requires all variables to have a type and that operand types in a dyadic expression be equivalent. Such equivalency is achieved by either declaring the variables involved to have the same type, or by explicitly casting one operand into the type of the other. Neither requirement is needed by the Bliss compiler so variable typing must be performed by the translator. As a notational convenience, variable type will be described as (*base level*) where *base* is C's <type specifier> and *level* is the level of indirection.

There are a number of ways for determining the base type of a variable from its use in the Bliss source code. For example, loop variables, field parameters and array indices must be integers. Variables found in BIND declarations and plits can have their types determined from initial values, and operands of floating point operators must be floating point numbers. Section 3.2 describes how to determine the structure type of variables mapped onto certain access algorithms. However, there still remain a large number of variables which have very uninformative declarations and are used with overloaded operators. With no other clues one must turn to the dyadic expression, and from the equivalency requirement try to deduce variable type. For example, suppose x is (integer 0), but y is unknown in the expression :

```
y <- .x;
```

A safe assumption would be that y is also (integer 0). When determining variable type using an assignment expression it is important to take into account the *expression type*. Expression type is the overall type of either the right hand side expression (RHS) or the left hand side expression (LHS). If the above expression is modified to:

```
y <- ..x;
```

where x is (integer 1) then y must be (integer 0), as it has been assigned to an expression of type (integer 0). In general the type of an expression is determined by:

1. The type of a known variable and its level of dereference. If the type of the known variable is (B_K L_K) then the expression type is (B_K L_K - EL_R) where EL_R is the explicit level of dereference of the RHS expression.

2. The type of any overriding operators. For example, in the expression : y <- FLOAT ..x;
even if x was (*integer* 1) then y must still be (*float* 0) because of the presence of the floating point operator.

Now, given the type of the RHS expression (B_R L_R) and the level of dereference of the LHS expression (L_L) the type of the LHS variable must be (B_R L_R + L_L). For more complex expressions one can perform a recursive analysis using operators and subexpressions for determining type.

When analyzing an assignment expression four situations may arise, depending on whether the LHS and RHS expression types can be determined. First, neither expression may have a known type, and no information can be gained. Secondly, both expressions may have a known type, and if they are equivalent no further analysis is necessary. However if the types are inequivalent then the RHS must be cast or the LHS variable must be declared a union of types. Ideally only the RHS has a known type as, through assignment, the LHS variable takes on a new value and perhaps a new type. In most cases the new type only makes sense if it is derived from the RHS expression type, although exceptions may occur where pointers to structures are encountered. Finally, if only the LHS has a known type then little can be determined from the expression, although as a last resort the RHS will be given its type. In truth, however, the action is meaningless and tends to lead to errors. The semantics of the assignment expression imply that the LHS variable takes on the attributes of the RHS, but not vice versa. For instance, suppose *X* is known and *Y* is unknown in the expression

x <- .y

Program semantics say that *X* takes on the value of *Y* and, implicitly, the type[1]. If one assumes that *Y* is the same type as *X* , and *Y* is used in later assignments the "known" type will be propagated to other variables. However, if the assumption is wrong then many variables could end up wrongly typed. Such an assumption in Bliss would not be safe as, for instance, pointer variables tend to be used to point to many different types. It may be safer to ignore the expression and hope some other clue could be found to type variable *Y* .

---

[1] Of course, in a rare instance the programmer may use the value in a completely different sense. Such hacks would be very difficult to deal with.

3.2. Structures from Access Algorithms

In section 2.2 it was shown that Bliss access algorithms in the form of equations (1) and (2) can be thought of as C member operators. Each invocation requires input arguments which supply the k_i's and the desired field j. In C, the k_i's and j are constants supplied by the compiler, but Bliss has no such constraints. In particular variable arguments can be used in the algorithm, making the conversion to C operations difficult, if not impossible. One can only assume that if the algorithm was intended as a structure member operator then its use will be consistent with this intent. A piece of data can always be treated as a structure if all input parameters are constants or, better still, the constant input parameters are defined in a macro name.

A C structure declaration requires a list of field names and their types, information required by the compiler to calculate the k_i's and to uniquely identify each field. Programmers are forced to label their fields by (hopefully) mnemonic names, rather than a number j as in equation (1). In a sense the Bliss input arguments correspond to the C field names, and if macro's are used the correspondence is one-to-one.

Suppose a Bliss program has all of its access arguments defined as macros. Then taking a data flow analysis approach one can determine the fields of a particular structure. For example, consider the code in figure 13.

```
ROUTINE CountBears(a, b) =
BEGIN   MAP member a, b;
        LOCAL member x, y;
        y <- x <- .a[big];
        y[average] <- .a[teddy] - .x[grizzly] + .a[bear];
        x[average] <- .y[average] + .b[teddy] + .b[bear];
END
```

Figure 13: Example for Data Flow Algorithm

Variables *a,b,x* and *y* have been mapped onto *member*, an access algorithm in the form of equation (2). All access arguments have been defined as macros, the names *big, teddy, average, grizzly* and *bear*. With the above information one can assume that the variables are structured. The problem remaining is to determine the C attributes necessary for the structure declarations. A data

flow approach to the problem is to follow the blocks of data represented by the "top level" variables *a, a[big]* and *b* . A top level variable is one that initially introduces a block of storage. A variable declared in an allocation declaration would be top level, as would be a parameter in a subroutine defined in a module separate from its calling program. The latter is necessary because the translation is working at a local level, without information from other modules. Notice this also includes fields of parameters, such as *a[big]*, whose origins are not always clear.

From the macros used directly with variables *a* and *b* one can deduce that *a* points to a structure with fields (big teddy bear) and *b* points to a structure with fields (teddy bear). Similarly variable *x* has fields (grizzly average) while *y* has fields (average). However, the first expression in figure 13 indicates that *x* and *y* are not top level variables and they simply point to whatever *a[big]* points to. Hence all fields associated with *x* and *y* should be associated with *a[big]*, which would then be a pointer to a structure with fields (grizzly average). The data flow algorithm is then:

1. Determine the set {S} of top level variables. Each element S_j is the first element in a subset of variables $\{V\}_j = S_j$.
   Thus the ith variable of the jth element in {S} is denoted v_{ij}.
2. Associate a potential structure with each element S_j in {S}.
3. For each assignment of the form v_{ij} <- x remove v_{ij} from the set $\{V\}_i$.
4. For each assignment of the form x <- v_{ij} include x in the set $\{V\}_i$.
5. Add all field names from expressions of the form v_{ij} [<field:name>] to the structure associated with S_j.

For routine *CountBears* the algorithm yields three distinct structures, namely: *S1*, with fields (*big, teddy, bear*); *S2*, with fields (*teddy, bear)* ; *S3*, with fields(*grizzly, average*).

Several questions have not been addressed by this algorithm. First, what if macro names are not always used? Data accesses such as .x[1, 18, 18] are not unknown in Bliss programs. However, as long as the input arguments are constant the algorithm will be correct. Unique names can always be generated by the translator.

Secondly, there is the question of completeness. Has the translator encountered all the fields associated with each structure? This question can only be answered if an entire program has been analyzed. In fact there are several problems associated with partial translation (module by module) as discussed in section 4.

The third question involves uniqueness. Are the structures generated unique or could some represent the same object? Three structures were generated from the example in figure 13 and variable *b* 's structure (teddy bear) seems to be a subset of variable *a* 's (big teddy bear). If variable *b* had the extra field, and if the fields with the same name had the same type then it could be concluded that *a* and *b* have the same structure. Life is never easy though, and the translator must hazard a guess as to whether or not the structures should be merged. In many cases programmers use overloaded field names, so the matching of a few field names is not enough to warrant merging. The question as to how much overloading occurs in a given Bliss program is one that can only be answered by the programmer. Thus either heuristics must be used or questions must be asked to determine how structures are to be merged. In the Bliss-to-C translator the user inputs a merge ratio at the begining of a run to help the system out. A merge ratio is the number of fields two structures should have in common, relative to the total number of fields of the largest structure. Thus a merge ratio of 2/3 means that if two thirds of the names in one structure are found in another then the structures should be merged.

Using a 2/3 merge ratio in the example of figure 13 and the typing algorithm of section 3.1 (defaulting to the operators + and - since no other information is available), the C declarations derived for subroutine *CountBears* would be:

```
struct S1
  {    struct S2 *big;
       int       teddy,
                 bear;
  }  *a, *b;

struct S2
  {  int grizzly,
         average;
  }  *x, *y;
```

Theoretically the ordering of fields in a C structure is unimportant; the details of implementation are taken care of by the compiler. On the other hand order is explicitly given by the Bliss programmer, as he must define exactly the position and size of each field. Is it necessary to preserve order in the translation? To answer the question consider the following code segment

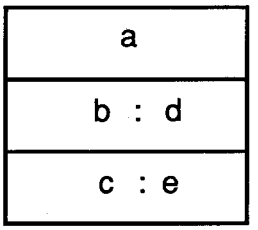
which operates on structure *y*.

```
STRUCTURE node[w] = [w] (.node +   .w)<0,36>;
MACRO a = 0$,
      b = 1$,
      c = 2$,
      d = 0$,
      e = 1$;
Map node x, y;

y[a] <- 5;
x <- y[b];
p <- .x[e];
q <- .y[c];
r <- .x[d];
```

y:

x:



The data flow algorithm will find *y* and *x* 's structures to have fields (a b c) and (d e) respectively. Since *c* and *e* represent the same memory location variables *p* and *q* must have the same value in the program. If the translator paid no attention to order, and flipped fields *d* and *e* in the structure declaration chaos would result. Preserving order is actually a simple matter for the translator. The fields need only be sorted by the argument which represents j in equation (1). Actually the problem of nested structures is more complex than just preserving order and will be dealt with more fully in section 4.1.

Finally, one must consider the case where the input arguments to an access algorithm are variables. Since C structure operators can only deal with constants it is necessary to turn to either the array member operator or an access function. An array member operator can be used if the following criteria are met:

* The access algorithm is in the form of equations (1) and (2), with the k_i's constant.
* The input parameters are all constant, with the possible exception of integer variables which correspond to the element indicies.

If the above conditions are true for a given variable then the task of the translator is to determine the type of the array elements. Element types can be determined as described in section 3.1, remembering that the index plays no part in type. For example the code

```
a[i, next] <- .c;
a[i, value] <- .d;
```

implies *a* is an array of structures with fields *next* and *value* having the same type as variables *c* and *d* respectively. In the implementation of the translator all structured variables were treated as simple structures unless variable access arguments were found. Then, if the array conditions were met a flag was set, which indicated the variable should be declared as an array. The size of the array may be difficult to find unless the variable is allocated in the module being translated. In the latter case the <size chunk> value could be used to determine range.

4. Results

An implementation of the algorithm discussed in section 3 was tested on several sources of data, including artificial examples, examples from [Wulf 71] and modules from a large CAD project. The success of the translation was measured by comparing the structures and types derived by the program to those derived through manual translation. Examples of the data are shown in appendix 1. On average only 78% of the variables were correctly typed. Many had several types, especially pointer variables, sometimes indicating the programmer took advantage of Bliss's untyped nature, sometimes indicating the failure of the translator to correctly merge structures. The problems were caused by several factors, as discussed below.

4.1. Deriving Structures

Appendix 1 contains the data structures found in a large Bliss module. Three sets of structures are shown, namely the original ones as described by the programmer, and two sets using different merge ratios. The structures are represented by a box containing several slots, one for each field. If a field has more than one label then each name will appear in the slot, along with its type.

As can be seen from the original data the assumptions made earlier did not always hold. There were a few instances where a field had more than one name or more than one type. The translator did not have any trouble coping with these violations though - it was overloaded field names that caused the most trouble.

Varying the merge ratios can change the results of the field collection algorithm drastically. Although ratios near 0 and 1 are clearly no good, with those in the middle it is hard to choose one as a winner. It seems that a ratio that works well for one structure fails miserably with another and correcting an under-merging of one structure would cause an over-merging of some others. Still, it would be nice to try and quantify the effect of the different ratios and see how they compare.

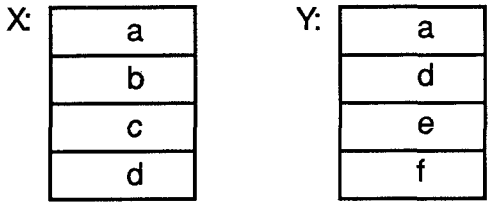
The effects of the various processes involved in the translation manifest themselves in the same way, namely to produce type unions, and it is impossible to discern the cause from the data alone. For instance, a union could be from under-merging, over-merging, or incorrect typing. However, the latter case should remain constant for different merge ratios and a union count can be used for comparison. A union count is calculated as follows:

* each field name with a type union scores 1,
* the total score for a structure is divided by the number of field names in the structure.

For the data in appendix 1 the union count is .25 and .30 for a merge ratio of 1/2 and 2/3 respectively. With this measure a lower score is better, and it would appear that a merge ratio of 1/2 works best. The numbers are somewhat misleading though, as the calculation lends equal weight to unions created through under merging and to those created through over-merging. Over-merging, however, produces far less unions than under-merging, as evidenced by figure 1 (all structures in appendix one merged together). Thus another measure is needed to correct the imbalance, such as the total number of structures created. The inverse of this number increases with over-merging, and if added to the union count may give a better number for comparison. For a 1/2 merge ratio the number is .375 while for 2/3 it is .366.

Do the numbers have any meaning? Intuitively, the 2/3 ratio produced a better solution than the 1/2 value, and this is what the measure quoted above predicts. The question, then, is how much faith can one put in two unweighted, somewhat arbitrary numbers added together? Not a great deal. To make the numbers more meaningful would require a weighting scheme, the weights being either statistically or theoretically derived. Whether or not this would be a useful thing to do is not really clear from the data. None of the ratios tried give a solution that outshines the others and a quantitative value could be more misleading than useful.

Why did structure merging fail as much as it did? The predominant reason was the fact that the information required to derive data structures is spread throughout an entire program. A particular module may not reference some of the fields associated with a structure, leaving its declaration incomplete. Since merging depends so strongly on the fields found, missing information can mean the difference between success and disaster. For example with structures X and Y (below) any merge ratio greater than 1/2 will fail to recognize them as incomplete parts of the same structure.



Lowering the ratio will help in this case, but this can be dangerous if overloading is encountered. The success of the algorithm depends on the coding style of the Bliss programmer.

Two things could be tried to fix up the algorithm. First, a more global analysis must take place. Good programming practice tends to build modular programs, and in Bliss this means spreading the necessary information around. The translator should make one pass over the entire program before attempting to analyze individual modules. Secondly, since the algorithm depends on coding style, more user input could be solicited. Exactly at what stage the input would be most effective is not yet clear. However, if the translator did some preliminary analysis and could present this to the user in a straight forward manner, much of the confusion would be avoided. Section 4.3 further discusses the interactive approach.

A second problem appears when nested structures are involved. Although not as fundamental as the one describe above it does lead to incorrect structures. The example of figure 14 illustrates the point.

```
STRUCTURE G[w,p,s] = [w] (.G + .w   )<.p, .s>;

STRUCTURE H[p,s] = [1] (.H)<.p, .    s>;

MACRO  list=1,18,18$,
       count=0,0,36,
       a=1,0,18$,
       m=18,18$,
       n=0,18$;

MAP    G y,
       H x;

         y[count] <- 4;
         x <- y[list];
         x[n] <- 5;
         x[m] <- 6;
          ...
         p <- .y[a];
         q <- .x[n];
```

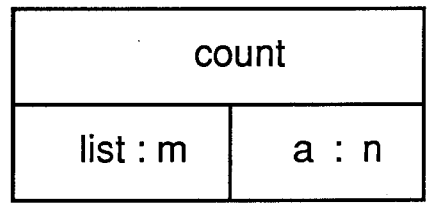


Figure 14: Example of Nested Structures

The code will be analyzed to determine that variable *y* is a structure *S1* with fields (*count,list,a*) and that field *list* is a structure *S2* with fields (*m,n*). Unfortunately nothing in the control statements link fields *list* and *m* or *a* and *n* . Using the algorithms of section 3 structure *S1* would look like:

```
struct S1
  {   int              count;
      struct S2
        { int m, n; } list;
      int              a;
  }
```

Of course, this means that variables *p* and *q* will not necessarily have the same value, yielding a bug in the translation. The problem is that the Bliss program has an implicit union of two structures with different field names. Only with a careful study of the macro and structure declarations along with the expression

```
x <- y[list];
```

can the problem be recognized. The expression says *x* points to the same block of storage as *y.list*, hence there may be a relation between the fields of *S1* and *S2*. The macro definitions along with the structure declarations indicate that the pairs (*list,m*) and (*a,n*) are names for the same field. To capture the true flavor of the structures in figure 14 would be difficult with C declarations, and it may be better to define *S1* as:

```
struct s1
  {   int count,
      list,
      a;
  }
```

Then an explicit cast could be done on the expression

```
x <- y[list];
```

to yield

```
x = (S2 *) &y.list;
```

4.2. Deducing types

With only partial success in the merging algorithm it is a little difficult to measure the success of the typing algorithm. For instance, it is possible to have a variable pointing to three separate structures, hence it requires a union of types, but if those three structures were correctly merged the variable would have a single type. In fact, when looking at a variable type one of five situations can arise, namely:

* the variable has a single, correct type,
* the variable has multiple types, but proper merging would yield a single, correct type,
* the variable has multiple types, some are correct while others are not,
* the translator was unable to find the variable type, and
* the variable is incorrectly typed.

The first case is simple, the algorithm worked. Similarly, in the second case the typing algorithm worked while the merging algorithm did not. Failure starts to become evident in the third case however, where incorrect types start appearing. Several factors contribute to this failure, as will be discussed shortly. Finally, in several cases the algorithm was unable to find a variable type. The problem is one of incomplete data rather than one with the algorithm.

In order to get some kind of quantitative estimate of the algorithm's success the following scoring scheme was employed:

* each correct type gets one point,
* a variable with multiple types due to incomplete merging gets one point,
* each variable with partially correct types scores a partial point, calculated by dividing the number of correct types by the total types deduced,
* wrong types score zero, and
* unknown types are ignored since they are caused by incomplete data rather than algorithmic failure.

The scores for each structure are added up and then divided by the total number of fields in the structure (subtracting fields of unknown type). The number yields the percentage of correct types. For the data shown in appendix 1 a merge ratio of 1/2 yields types that are 79% correct, while a merge ratio of 2/3 yields 76%.

Several factors contributed to incorrect typing. First, in any given Bliss module there may not be enough information to start the ball rolling. In order to use the data flow algorithm initial data values must be known. Such values are found in BIND declarations and plits, or in assignments to constants. As is often the case though, initializations may be done in a module separate from the rest of the program. When the translator tries to tackle other modules the majority of assignment expressions have unknown LHS and RHS expression types.

In order to overcome the problem one could approach the translation in a more global sense. Subroutines could be processed in the order of their calling sequence, ensuring the types of function arguments are known before they are passed. However a data flow algorithm of such magnitude would be very hard to implement and, for large Bliss programs with many modules, would be prohibitively inefficient.

A less involved solution would be to make an information gathering pass over each module before the translation begins. Such a pass would result in a partial symbol table containing the types derived from initializations of top level variables. Although this would not necessarily cure the problem it could help to the point where minimal human interaction would be necessary.

A second problem is encountered when a variable is assigned a value of zero. Quite often a value of zero represents a null pointer to some structure. Unless another expression involving *x* is encountered which resolves the conflict the variable type must be labeled *ambiguous*. Pointer arithmetic causes similar problems, and expressions such as:

```
x <- .x + 1;
```

can be looked at as either incrementing a counter or moving through an array.

The implementation treated all ambiguous variables as untyped, until a more positive indication of type turned up. If, at the end, there were still such variables their type defaulted to integer. Only 50% of the ambiguous variables found had successful resolutions. The others were variables that were initialized in other modules, and the disambiguating information just wasn't present in the code being analyzed.

Third, recall that the typing algorithm is based on data flow analysis. Unfortunately, the analysis proceeds only at a local level. Each subroutine is processed in sequence, but there is no inter-routine analysis. Thus, one may run into trouble with global variables. For instance, consider the code shown in figure 15. Using the data flow algorithm the routines will be processed in the order of 1st, 2nd, 3rd. Variable x is global to each of these routines, and is assigned values in both 1st and 2nd. The value of x is used in the routine 3rd. Suppose variables y and z have different types. The heuristic will assign each type to x, but the current type when processing routine 1st will be the type of z. Thus variable r receives the type of z, but in reality, because of the order in which 2nd and 1st are called, its type should be that of y. Again, the best way to handle a situation like this would be to analyze the subroutines in the order of their execution. As it is there is no way of detecting the current type of a variable whose value was obtained outside of the routine being analyzed.

```
OWN x;
ROUTINE 1st
 ...
 x <- .y;
 ...
ROUTINE 2nd
 ...
 x <- .z;
 ...
ROUTINE 3rd
 ...
 2nd(p);
 1st(p);
 r <- .x;
 ...
End;
```

Figure 15: Example of problems in global analysis

Finally, there is a problem caused by an implicit union of types in Bliss's 36 bit word. Suppose, for example, argument *node* is passed to a routine and processed by the statements in figure 16. The typing of *node* , a simple application of the field collection algorithm, presents no serious problems, but the typing of its fields does.

```
edgelist <- node[edge];
...
WHILE (...) DO
 ( .edgelist <- GETSPACE(1);
   edgelist <- ..edgelist;
   edgelist[value] <- .edge2;
   edgelist <- edgelist[next];
   ...);
```

Figure 16: Example of incorrect variable typing

In the example local variable *edgelist* is assigned one of the fields of *node* and the field is a pointer to a list, or (list 1). Since *edgelist* is assigned to the address of this field, its type becomes (list 2). Statement 2 yields no new information, while statement 3 requires *edgelist* to have a new type, namely (list 1). The next two statements introduce the fields belonging to list, *value* , and *next* . It is the last statement though, that creates a problem. As this is the only statement in the routine that uses the field *next* , its type is not known. The type of *edgelist* is known though, and by default the algorithm incorrectly assumes *next* must have type (list 0) to ensure valid expression. In reality *edgelist* should go back to its original type (list 2) and *next* should have type (list 1).

The problem here is that Bliss's 36 bit word can contain one 36 bit byte pointer or two 18 bit addresses and therefore has a dual identity. On one hand *edgelist* is treated as a pointer to an 18 bit address while on the other hand it is treated as a pointer to a structure with two 18 bit fields. *Edgelist* is implicitly a union of two types. The heuristics used in this thesis are not powerful enough to recognize the subtle use of *edgelist.* However, it will be shown that through program transformation the problem can be diverted from the translator and easily corrected by hand.

4.3. Program Improvement

Metaprogramming [Cameron 84] is the art of writing programs to manipulate other programs. By this definition a translator is one type of metaprogram, inputting source code in one language and transforming it to another. Cameron and Ito have developed a system to automatically produce tools for metaprogramming, based on the grammar of the target language. Using their GRAMPS generator it was possible to produce a complete set of tools for manipulating Bliss programs.

With a good metaprogramming system it is conceivable that one can go beyond mere translation, and get into the realm of program improvement. Program improvement, as discussed in section 1.1, can be achieved by upgrading the quality of the source code. One way to improve quality is to search for examples of data manipulation in a program, and transform these into a more abstract form. One must ask, though, why bother with an interactive approach to program transformation? After all, the point of a translator is to transform code with a minimum of human effort. The art of program manipulation has a long way to go before quality improvement can be automated. However, there are three important reasons why one should attempt a source-to-source transformation before the actual translation. First, any improvement in the code quality will have beneficial effects on program maintenance. Maintainability, of course, plays a large part in the style of translation adopted. Secondly, a user directed transformation beforehand may reduce the number of problems encountered by the translator. For instance, it was shown in section 4.2 that when the type of an input parameter is unknown the type of some of its fields may be deduced incorrectly. It will be shown in the next example how, through a metaprogramming transformation, one such problem was corrected.

Finally, given a metaprogramming system (such as GRAMPS), the programs required to produce various specific transformations require very little time to implement. In the following example the metaprogram was written in less than an hour.

Recall the code in figure 16 which led to a variable of incorrect type. The last four expressions represent a list building algorithm. Initially variable *edgelist* points to the *next* field of the last item of a list. The four expressions allocate a new element, initialize it, and update *edgelist*. In a program that fully supports data abstraction the list building code would be encapsulated in a function. For example, the function definition and the statements in figure 17 would replace figure 16.

```
FUNCTION AppendEdge(x, edge) =
BEGIN  LOCAL list x;
    .x <- GETSPACE(1);
    x <- ..x;
    x[value] <- .edge;
    x <- x[next];
    return(x);
END
```

Figure 17: Abstraction of list building code

A metaprogram, slightly generalized, to produce the desired transformation would look for an expression list of the form

```
.x <- GETSPACE(1);
x <- ..x;
x[value] <- <expression>
x <- x[next];
```

with, perhaps, intervening expressions which do not affect the value of x, and replace them with the expression:

```
x <- AppendEdge(x, expression);
```

AppendEdge is a simple function that could be translated by hand.

A metaprogram to perform the above transformation was written and yielded surprisingly good results. Careful scrutiny of the code by hand turned up three such expression sequences in a particular module, while the metaprogram correctly found and transformed all four.

Although there are good reasons why one should adopt a translation system as presented above, there is also one major reason why one shouldn't. If the user is not familiar with the program to be translated (let alone Bliss itself) then it may be difficult to identify useful transformations. Any time saving the metaprogramming system may yield could be insignificant compared to the time needed to achieve a minimal understanding of the code.

5. Summary and Conclusions

The thesis has presented arguments as to why a source-to-source translation system should emphasize the quality of the end product. Code readability and maintainability are of fundamental importance when translating production systems. As there are potentially an infinite number of semantically correct solutions to any translation one should use program quality metrics to try and find a reasonable one. A complete translation system represents several years of work, hence it was not possible here. Instead, an attempt was made at deducing data primitives in an untyped, unstructured language, directing the search by means of a simple quality metric. It was important to use this metric to try and separate out, from many possible solutions, one that reflected the original data primitives.

An example of a translation system was provided by the first phase of a Bliss-to-C compiler. Both are block structured languages, but have radically different data structuring mechanisms. Bliss is a language that implements access algorithms as its data structures. As such, it is untyped and unstructured. It was necessary, then, to employ a set of heuristics to deduce a higher level interpretation of the data. The thesis was broken into two parts, namely deducing data type, and deducing data aggregates. The success of the translation was measured by counting the number of type unions introduced as a result of the translation.

The simple heuristic used to obtain data types was 78% correct. This number was arrived at by comparing the results of the translation to the original data structures To produce better results would require substantially more effort in terms of data flow analysis; an entire program would have to be analyzed following the order of execution. The problem arrises because a great deal of information is left out of the Bliss code, information that is really locked in the head of the programmer, with parts spread throughout a program. Information that ties a lot of the little pieces together, and eliminates ambiguities cannot be found by a simple, local analysis, as presented here.

Other problems arise because the metric used here was too weak and ad-hoc in nature. Often, varying the merge ratio did not make any statisical difference to the value of the metric, while it was obvious that a significant change had taken place in the data aggregates themselves. The changes were not necessarily better or worse in terms of quality, but such weak results indicate a need for better direction.

If one abandons the idea of an entirely automated translator and takes a more interactive approach the results improve substantially without the need to write highly complex programs. Using the GRAMPS [Cameron 84] system for producing software tools it was found that, with a little effort on the user's part, a more abstract form of a Bliss program could be found. The transformed Bliss code was much more suitable for translation than the original.

The work done here has shown that the use of quality metrics can help direct a source-to-source translation. The number of sets of data primitives that can be derived from a program is large, but the knowledge of what is reasonable and what isn't can help in the pruning. Searching for a suitable quality metric must continue though, to find one that more accurately measures the difference between one set of primitives and another. Since the data aggregates are symbolic, rather than numeric, in nature it is probably better to look for some symbolic means of describing an aggregate, and derive a metric that measures the difference beween two descriptions.

The results here have also indicated that syntactic knowledge of a language is not necessarily enough to deduce data primitives in an unstructured language. This is not surprising, since building a structured program requires much more input on the programmer's part. It is probably necessary to capture more pragmatic knowledge, a much deeper understanding of the source language, in order to cope with the problem adequately.

Finally, it should be concluded that total automation of a translation system is not practical, that more time should be spent on building an interactive system in which the user can fill in the missing gaps and help resolve ambiguities. The knowledge required by the program is readily at the programmer's finger tips. It may require relatively little effort on his part to impart this information, as the example using the metaprogramming system shows.

Appendix 1

The following appendix shows, in detail, the data structures from the Bliss program example used throughout the text. It is broken into three parts, the first showing the original data aggregates, derived by hand, the second showing the aggregates derived with a merge ratio of 1/2, and finally the aggregates derived with a merge ratio of 2/3. Each aggregate is described by a list of boxes. Each box has one or more names and types. A single name and type implies no unions, a single name and several types implies a union of types, and several names and types implies the field represents more than one entity. In some cases the type is denoted (), meaning that the translator was unable to derive a type.

Tree:

dfidnum (int)	
dfdad (tree 1)	
dfVTptr (activity 1) (outnode 1)	
dfsons (tree 1) (list 1)	
dfcros (tail 1)	doC (tail 1)
dffwrd (forward 1)	doF (forward 1)
doLrev (Lrevlist 1)	doL (Llist 1)

List :

llval (tree 1)
llnxt (list 1)

Tail:

nlnxt (tail 1)
dotail (int 0)
dohdlst (head 1)

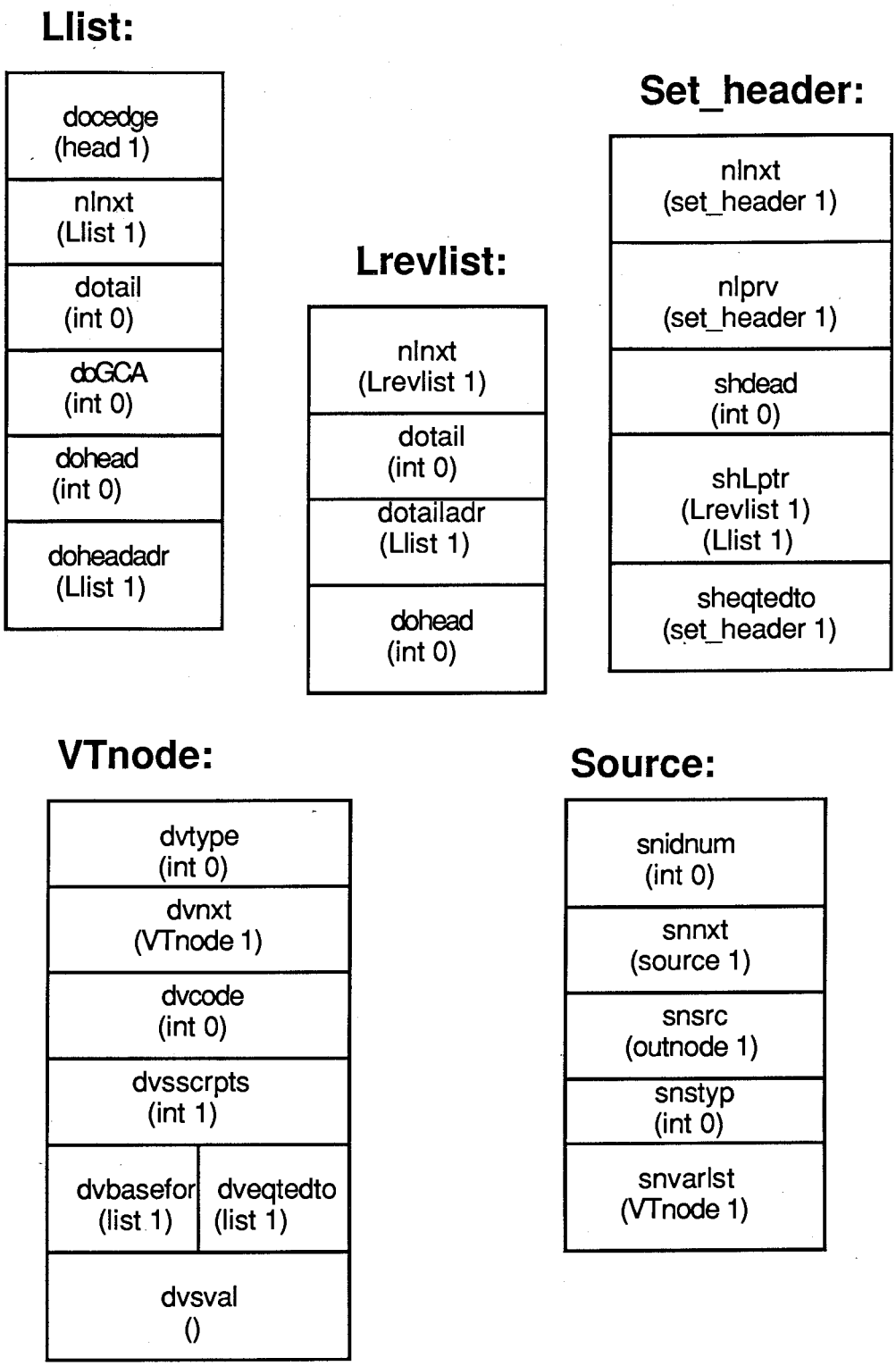
Head:

nlprv (head 1)
nlnxt (head 1)
dotailadr (tree 1)
doheadadr (tree 1)
dohead (int 0)

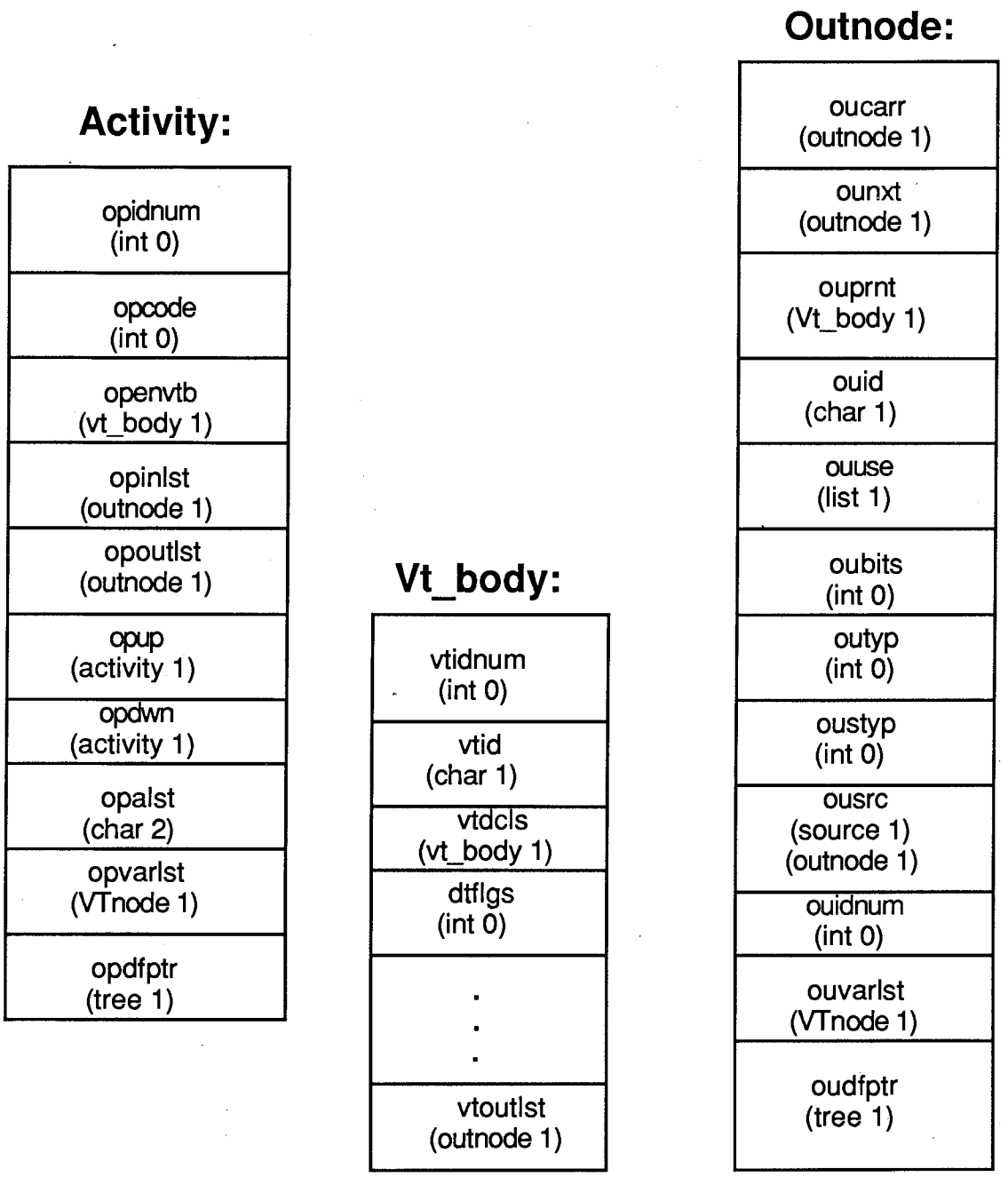
forward:

nlnxt (forward 1)
dofhead (int 0)
dofheadadr (tree 1)

Original Data (a)



Original Data (b)



Original Data (c)

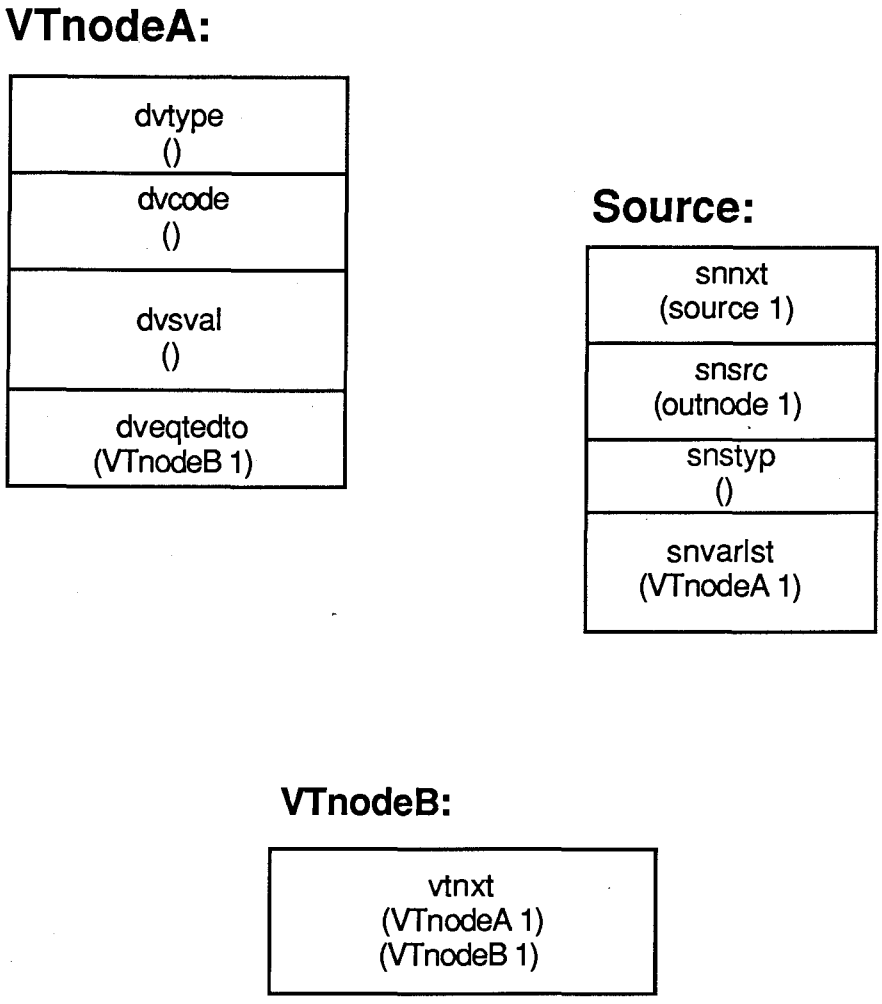
Tree:

dfwrd ()	
dfidnum (int 0)	
dfdad (tree 1)	
dfVTptr (activity 1) (vt_body 1)	
dfsons (tree 1) (composite 1)	
dfcros (composite 1)	doC (composite 1)
dffwrd (composite 1)	doF (composite 1)
doLrev (composite 1)	doL (composite 1)

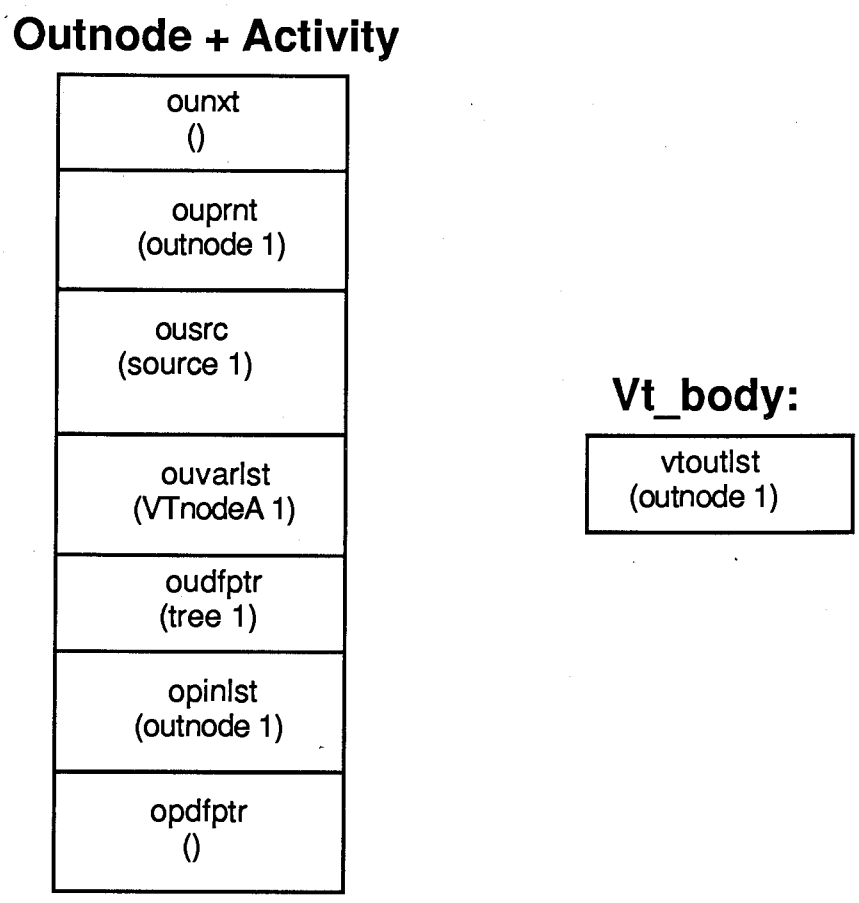
forward:

nlnxt (forward 1)
dofhead ()
dofheadadr (composite 2) (composite 1)

Merge Ratio 1/2 (a)

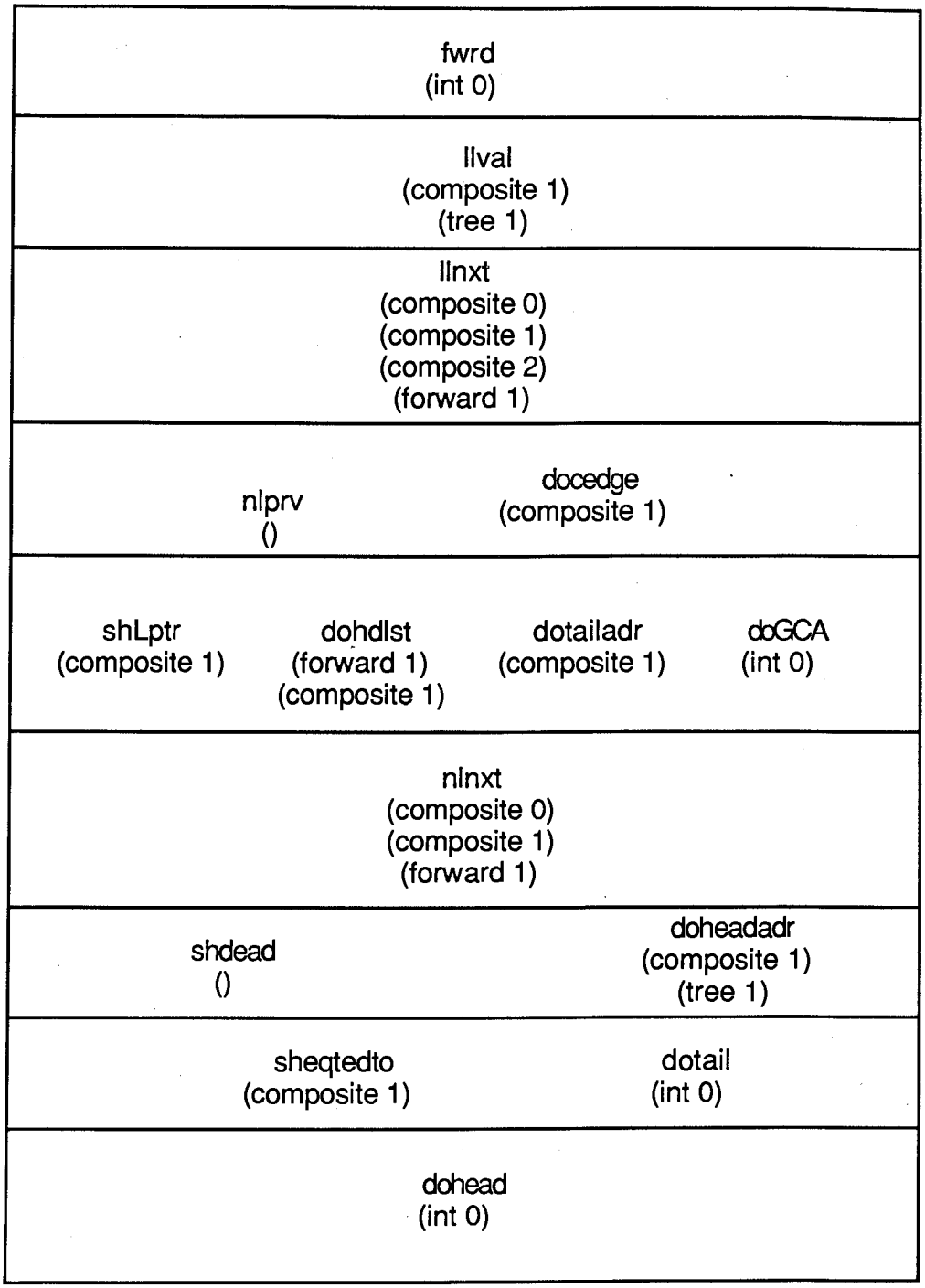


Merge Ratio 1/2 (b)



Merge Ratio 1/2 (c)

Head+List+Set_header+Tail+Llist+Lr evlist



Merge Ratio 1/2 (d)

Tree:

dfidnum (int 0)	
dfdad (tree 1)	
dfVTptr (activity 1) (outnodeA 1) (outnodeB 1) (outnodeC 1) (VT_body 1)	
dfsons (tree 1) (list 1)	
dfcros (list 1)	doC (composite 1)
dffwrd (list 1)	doF (composite 1)
doLrev (composite 1)	doL (composite 1)

List :

llval (tree 1) (list 1) (set_header 1)
llnxt (list 0) (list 1) (composite 2) (set_header 1) (forward 1)

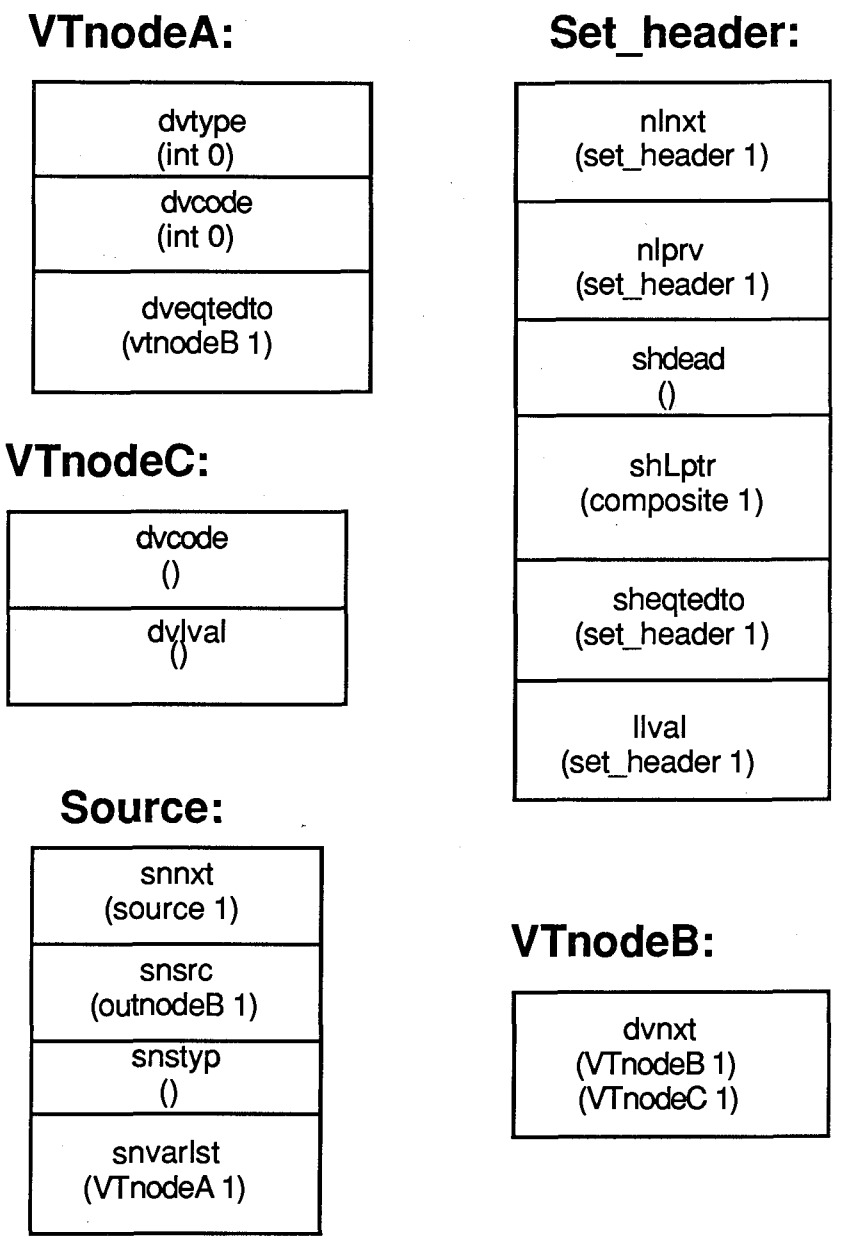
forwardB:

nlnxt (composite 1) (list 2)
dofheadadr (tree 1)

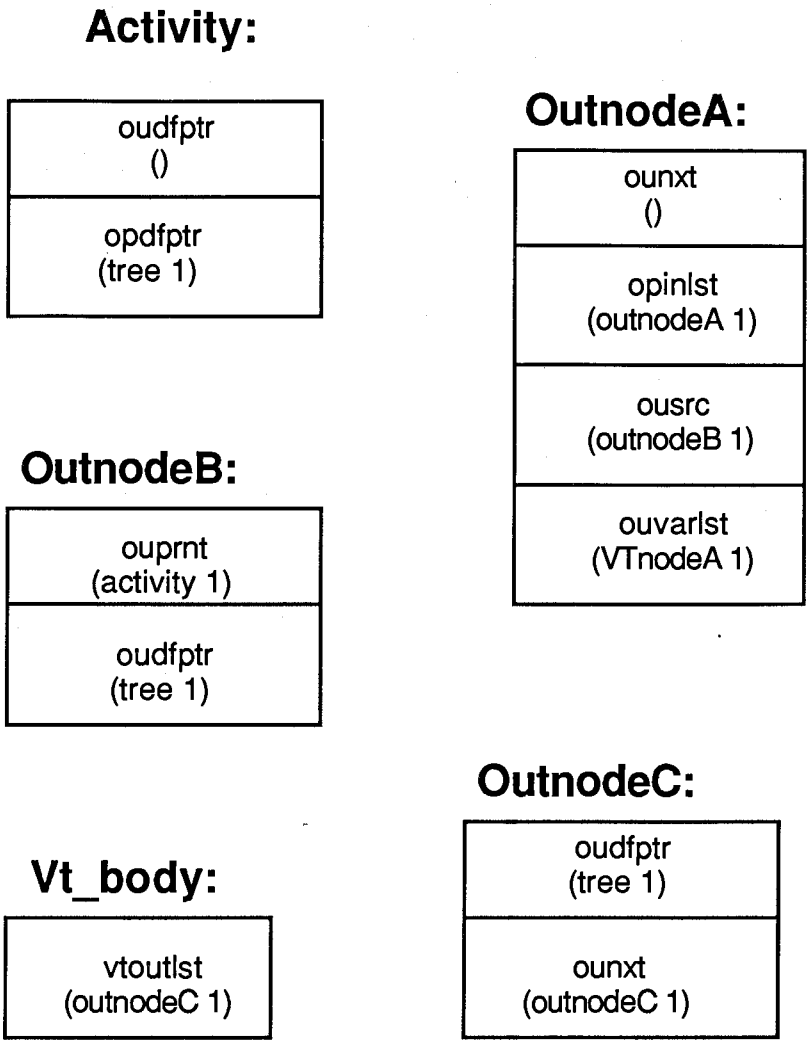
forwardA:

dofhead ()
dofheadadr (tree 1)

Merge Ratio 2/3 (a)

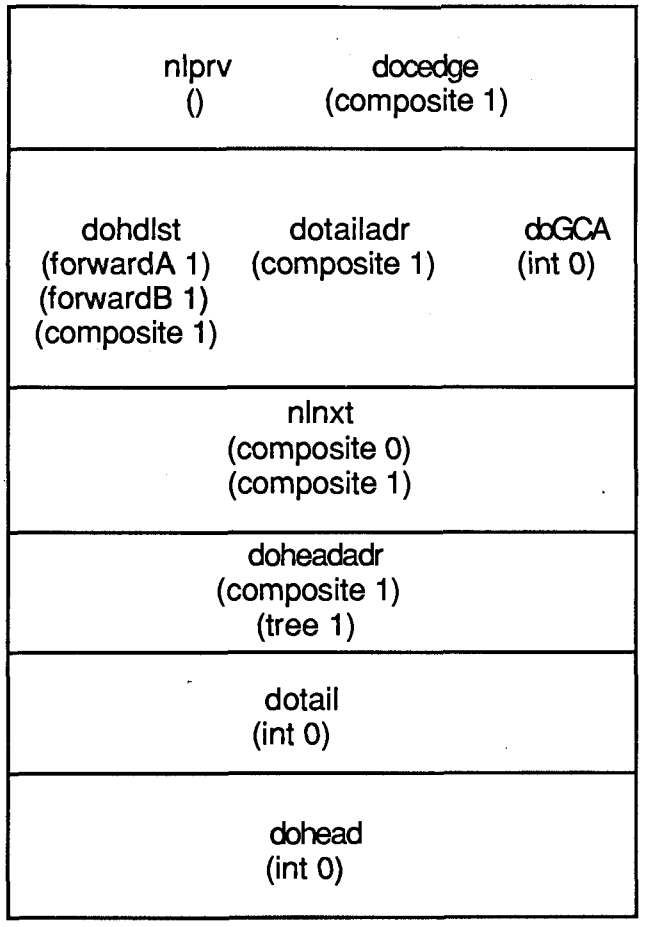


Merge Ratio 2/3 (b)



Merge Ratio 2/3 (c)

Head+Tail+Lrevlist+Llist:



Merge Ratio 2/3 (d)

References

[Albrecht 80] Albrecht, P.F., P. Ip, B. Krieg-Bruckner, P.E. Garrison, S.L. Graham, and R.H. Hyerle.
Source-to-Source Translation: Ada to Pascal and Pascal to Ada
In *Proc. of the ACM-SIGPLAN Symposium on the Ada Programming Language*. ACM, New York, December, 1980.

[Bergland 81] Bergland, G.D.
A Guided Tour of Program Design Methodologies.
IEEE Computer 14(10), October, 1981.

[Brown 81] Brown, J. C., M. Shaw.
Toward a Scientific Basis for Software Evaluation.
In A. Perlis,F.Sayward, M. Shaw (editors), *Software Metrics: An Analysis and Evaluation*, pages 19-41. MIT Press, Cambrige, 1981.

[Cameron 84] Cameron, R. D., and M. R. Ito
Grammar-Based Definition of Metaprogramming Systems.
ACM Transactions on Programming Languages and Systems 6(1):20-54, January, 1984.

[Freak 81] Freak, R.A.
A Fortran to Pascal Translator.
Software - Practice and Experience 11(7):717-732, July, 1981.

[Griffiths 79] Griffiths, M.
Translation Between High Level Languages.
In P.J. Brown (editor), *Software Portability*, pages 106-113.
Cambridge University Press, New York, 1979.

[Halstead 77] Halstead, M.H.
Elements of Software Science.
Elsever North Holland Inc., 1977.

[Harrison 82] Harrison, w., K. Magel, R. Kluezny, A. DeKock.
Applying Software Complexity Metrics to Program Maintenance.
IEEE Computer 15(9), September, 1982.

[Jackson 75] Jackson, M.A.
Principles of Program Design.
Academic Press, London, 1975.

[Kernighan 78] Kernighan, B.W., and D.M. Ritchie
The C programming Language.
Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.

[Linden 76] Linden, T.A.
The Use of Abstract Data Types To Simplify Program Modifications.
ACM SIGPLAN Notices 8(2):12-23, March, 1976.

[Peck 78] Peck, J.E.L., and G.F. Schrack.
The Portability of Quality Software: Experiences with BPCL.
In P.C. Hibbard, S.A. Schuman (editors), *Quality Software.*
Elsvier North-Holland, Inc., New York, 1978.

[Rowe 81] Rowe, L.A.
Data Abstraction from a Programming Language Viewpoint.
SIGART Newsletter (74):24-35, January 1981.

[Wulf 71] Wulf, W.A., D. Russell; A. Habermann
Bliss Reference Manual.
Technical Report, Carnegie-Mellon University, 1971.