SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(11), 1001–1023 (NOVEMBER 1994)

Efficient Binary Transfer of Pointer Structures

ian toyn and alan j. dix
Department of Computer Science, University of York, Heslington, York YO1 5DD, U.K.
(email: ian@minster.york.ac.uk,alan@minster.york.ac.uk)

SUMMARY

This paper presents a pair of algorithms for output and input of pointer structures in binary format. Both algorithms operate in linear space and time. They have been inspired by copying garbage collection algorithms, and make similar assumptions about the representations of pointer structures.

In real programs, the transfer of entire pointer structures is often inappropriate. The algorithms are extended to transfer partitions of a pointer structure lazily: the receiver requests partitions when it needs them.

A remote procedure call mechanism is presented that uses the binary transfer algorithms for communicating arguments and results. A use of this as an enabling mechanism in the implementation of a software engineering environment is discussed.

key words: Pointer structure Lazy transfer Distributed system Inter-process communication Remote procedure call Cadiz

THE PROBLEM

Many programs build data structures that consist of pieces of primary memory (often called records, structures or nodes) linked by memory addresses (often called pointers or references). Such linked data structures, or *pointer structures*, can be used to represent linear lists, trees, directed acyclic graphs, and unrestricted cyclic graphs. Pointer structures can be built, traversed, and revised, and finally the space they occupy can be reclaimed for reuse.

The problem is how to transfer a pointer structure from one process to another, either via secondary storage or more directly via an inter-process communication stream. The requirements on a solution are that it be able to cope with an arbitrary pointer structure (maintaining any sharing and cycles), and that the original pointer structure be left in its original state.

The links in the receiving process' copy of the pointer structure will almost inevitably be represented by different primary memory addresses to those in the original pointer structure, simply because primary memory at the same addresses is unlikely to be free. The major difficulty is in the maintenance of the links. Whether the transfer takes place via secondary storage or via an inter-process communication stream makes no difference.

CCC 0038–0644/94/111001–23

Received 22 November 1993
Revised 23 June 1994

MOTIVATION

Language processing tools, such as compilers and interpreters, infer interesting properties from the programs that they process. For example, when a type error is detected in an expression, the types inferred from neighbouring expressions are of interest, as they might help to suggest a correction. As another example, someone browsing a program might wish to discover which definition is referred to by a particular reference, or wish to trace all uses of a particular definition. Regular expression search strings are often used in this task but they can easily mislead, for instance when names are prefixes of other names, and especially when names are overloaded. It can be irritating for a programmer to know that a compiler or interpreter has computed needed information but won't reveal it.

Proper provision of such facilities involves presenting the programmer with a view of the program and the ability to point to an expression and ask questions about it. Adding this functionality to a compiler or an interpreter would complicate the existing tool unattractively. Good engineering practice demands separation of concerns, for example as provided by UNIX between separate processes. The proposed facilities should be implemented by separate tools.

One approach would be for the new tools to duplicate the syntactic and semantic analyses of the compiler or interpreter, effectively recomputing the needed information. Besides the inefficiency, such duplication raises the further problem of how to ensure that the duplicates are equivalent.

Another approach would be explicitly to communicate the annotated abstract syntax tree representation of the program, as constructed by a compiler or interpreter, to the new tools. If such explicit communication could be achieved efficiently, both in terms of execution time and without significant quantities of new code having to be written, then many benefits should follow. The ability of new tools to start from existing data structures should increase programmer productivity, as the new tools will be easier to produce. The separation and the avoidance of duplication should result in better implementations and hence should increase confidence in the correctness of the tools. Moreover, this method of integrating tools should allow greater flexibility regarding extension and modification of toolsets.

This was our motivation for developing algorithms for transferring data structures between processes. They would be applicable in other contexts, notably in distributed systems, and in interfacing alternative code generators to compilers.

In the next section, some terminology is established and some assumptions made that simplify the following presentation. The algorithms were inspired by a copying garbage collection algorithm, and so this is explained before the new algorithms. Examples are given to illustrate the effects of the algorithms. The paper then returns to the motivating theme of software engineering environments, showing how a set of tools for manipulating Z specifications was realized by using remote procedure calls between tools, with the arguments and results of those calls transferred using the new algorithms.

ASSUMPTIONS AND TERMINOLOGY

The data to be transferred is assumed to be stored in *cells*. A cell has a sequence of *attributes*. Each attribute is either a *literal* data value, such as a number, or a *pointer* to a cell (that cell's memory address). Given a pointer to a cell, it must be

possible to determine the size of that cell and the offsets of all of its pointer attributes. This can be achieved in several different ways:

(i) explicitly, by a *header* containing the size of the cell (or equivalently the number of attributes, if the attributes are of equal sizes) and a bitmap in which 0s and 1s represent literal and pointer attributes in order
(ii) by a smaller *header* encoding one of a collection of known types of cells, where all cells of a particular type have the same predetermined sequence of attributes
(iii) from the address value of the pointer, by having arranged for all cells of a particular predetermined type to reside within particular address ranges in memory.

The algorithms are expressed in a manner that does not commit to any of these ways of representing cells, but the illustrative examples assume way (ii).

All cells must be large enough to contain one pointer attribute. This is to allow for the storage of a *forwarding address* by the output algorithm to a copy of the original cell. These forwarding addresses enable the retention of sharing within pointer structures. Normally no extra storage is required, as a forwarding address overwrites an existing first attribute, whose value is subsequently restored.

In addition to cells, there is a set of *root pointers*. These do not lie within cells, but act as entry points into the pointer structures. It is assumed that the data to be ouput is the pointer structure referred to by a single root pointer; cells that are not linked into this particular pointer structure are not to be output.

When cells are transferred between processes they almost inevitably arrive at different addresses in memory from those from which they were copied. The major task for the algorithms is to *relocate* new pointers so that they point to corresponding cells in the new structure.

It is assumed that the bit patterns representing headers and literals remain the same across the transfer. For a distributed system, this assumption implies that the distributed system be a homogeneous one, i.e. one in which all the processors are of the same type. (This assumption simplifies the presentation; it isn't a requirement.)

COPYING GARBAGE COLLECTION

A *garbage collector* is an algorithm that reclaims the space occupied by inaccessible cells. The reclaimed space becomes available for the subsequent allocation of new cells. A *copying garbage collector*[1] copies all accessible cells into a new area of memory. The entire old space is then discarded, thus effecting the reclamation of the space occupied by inaccessible cells. Copying garbage collectors are popular relative to other garbage collectors for several reasons: their time cost is proportional to only the number of accessible cells, rather than to the size of memory; they compact the accessible cells into fewer pages of virtual memory; linked cells are likely to be copied as neighbours, improving so-called locality; and fragmentation is avoided due to the compaction.[2]

The new algorithms were inspired by the realization that a copying garbage collector satisfies some of the requirements for transferring pointer structures between processes: it succeeds in copying pointer structures (though not between processes), relocating pointers appropriately and retaining sharing, but it damages the originals.

The similarities between the new algorithms and a copying garbage collector are such that it is worth while explaining the copying garbage collection algorithm first.

Let the area of memory containing the original cells be called OLD SPACE, and the area containing the copies be called NEW SPACE. It is simplest to assume that these spaces are each contiguous areas of memory, though they need not be. Let pointer be the type of references to cells, and deref(p) be the cell referenced by the pointer p.

The garbage collector begins by copying those cells that are immediately accessible from the root pointers. (The copy procedure follows the gc procedure below.) It then sweeps through the copied cells, copying those cells that are directly reachable from them. Forwarding addresses are used to maintain sharing.

```
gc( ) {
    for each root pointer r,
        copy(r);
    for each cell c in NEW SPACE in order,
        for each pointer attribute p in c,
            copy(p);
            overwrite p in c with the forwarding address
            from deref(p);
}

copy(p: pointer) {
    if deref(p) has not yet been copied,
        copy deref(p) to the next free cell, q, in NEW SPACE;
        alter deref(p) to mark it as having been copied
        and to contain a forwarding address to the copy q;
}
```

Note that copy allocates the next free cell from a contiguous free space, hence appending the copy onto the end of previous copies of cells. Thus the sweep of the copied cells allocates more cells for itself to sweep; the sweep terminates when all accessible cells have been copied. (The sweep technique, first described by Cheney[3], avoids recursive graph traversals that would require an auxiliary stack.)

Imagine running a variant of the copying garbage collector in which instead of considering all root pointers, only the pointer to the structure to be transferred is considered. The resulting contents of NEW SPACE would be exactly the pointer structure to be transferred, nicely compacted. Pointer relocation could be achieved by replacing each pointer by the offset of the addressed cell in NEW SPACE on output and by adding the base address of the receiving space to this offset on input. (If either space is not contiguous then relocation becomes only a little more complicated.) Unfortunately, although the garbage collector would be of great assistance in transferring the data, it would have the unacceptable consequence of leaving forwarding addresses scattered over the OLD SPACE.

THE NEW ALGORITHMS

Outline

The output algorithm to be presented uses two spaces, referred to as FROM SPACE and PATCH SPACE. (These can be the same memory as OLD SPACE and NEW SPACE, respectively, used by a copying garbage collector.) It consists of a sequence of three phases.

1. The data to be transferred is copied out of FROM SPACE and compacted into PATCH SPACE, leaving forwarding addresses in FROM SPACE that were used to retain sharing.
2. From PATCH SPACE it is written to the output stream, relocating pointer values to stream offsets as they are written.
3. Finally, the information in PATCH SPACE (which differs from that which would be produced in NEW SPACE by a copying garbage collector) is used to restore the original contents of FROM SPACE.

The input algorithm is relatively simple, and operates as suggested above. The data is read from the input stream into a space referred to as RECEIVE SPACE.

The following types are assumed to be defined.

from_pointer	a subtype of pointer, constrained to reference cells in FROM SPACE.
offset	The type of stream offsets (natural numbers).

Output algorithm

The output algorithm is encoded in the output routine as follows. It has three auxiliaries: copy, relocate_out and refresh. The copy routine is essentially the same as that in the garbage collector: only the names have been changed. Note that code in the relocate_out routine that is enclosed in exclamation marks !like this! is needed only if PATCH SPACE is not contiguous.

```
output (o: stream, r: from_pointer) {
    initialize PATCH SPACE;
-- Phase 1, FROM-to-PATCH:
    copy(r);
    for each cell c in PATCH SPACE in order,
        for each pointer attribute p in c,
            copy(p);
-- Phase 2, PATCH-to-STREAM:
    for each cell c in PATCH SPACE in order,
        write c to o, but for each of its pointer
        attributes p, substitute relocate_out(p),
        leaving c unchanged;
-- Phase 3, PATCH-to-FROM:
    refresh(r);
    for each cell c in PATCH SPACE in order,
        for each pointer attribute p in c,
            refresh(p);
}
```

```
copy(p: from_pointer) {
    if deref(p) has not yet been forwarded,
        copy deref(p) to the next free cell, q, in PATCH SPACE;
        alter deref(p) to mark it as having been copied
        and to contain a forwarding address to the copy q;
}

relocate_out(p: from_pointer) returns offset {
    get the forwarding address f from deref(p);
    return f – base address of !referenced chunk of! PATCH SPACE
        !+ sum of sizes of any preceding chunks of PATCH SPACE!;
}

refresh(p: from_pointer) {
    if deref(p) is still a forwarded cell,
        overwrite the forwarding information with
        data taken from the forwarded copy;
}
```

Note the differences between the code of phase 1 and the garbage collector's gc operation: there is only one root pointer, and the pointers in cells in PATCH SPACE are not overwritten—those cells are exact copies of the original cells.

Input algorithm

The input algorithm is encoded in the input routine as follows. It has one auxiliary: relocate_in. Note that the code enclosed in exclamation marks !like this! is needed only if RECEIVE SPACE is not contiguous.

```
input(i: stream) returns pointer {
    initialize RECEIVE SPACE;
    for each cell c in i,
        read c from i;
        for each pointer attribute p in c,
            p := relocate_in(p);
    return a pointer to the first cell read;
}

relocate_in(p: pointer) returns pointer {
    return p + base address of !indexed chunk of! RECEIVE SPACE
    !– sum of sizes of any preceding chunks of RECEIVE SPACE!;
```

Example

The cyclic graph in Figure 1 will be used to illustrate the effects of the three phases of the output algorithm. The rectangles and arrows denote cells and pointers, respectively. Each cell has a header containing a value according to the following code: C for a cell containing two pointers (C for cons, as in LISP), and N for a number cell. The addresses of the cells are denoted by the labels *w*, *x*, *y* and *z*. This pointer structure might represent the infinite sequence [19, 89, 19, 89, 19, 89, …

EFFICIENT BINARY TRANSFER OF POINTER STRUCTURES

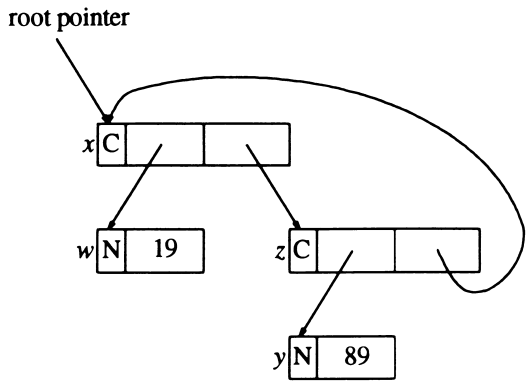

Figure 1.

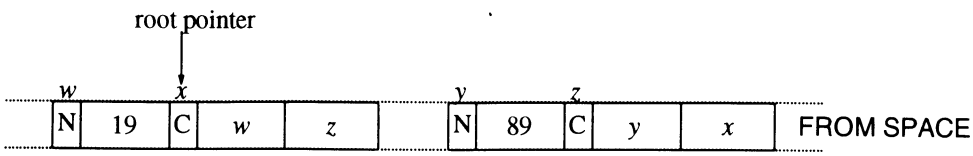

Figure 2.

Laid out in linear storage it could look as in Figure 2. The cells might not be contiguous, and the cell pointed to by the root pointer need not be first. The other pointers are denoted by the address labels *w*, *x*, *y* and *z*. This linear storage is the FROM SPACE for the algorithm.

The first phase copies all cells accessible from the root pointer to a compact region of PATCH SPACE. The resulting cells in PATCH SPACE are exactly the same as the original cells in FROM SPACE—even the pointers still refer to original cells in FROM SPACE. The cells in FROM SPACE have to be overwritten with forwarding addresses to preserve sharing. This overwriting is done by the copy routine. It overwrites the first attribute of each cell with a pointer to the copy of that cell in PATCH SPACE, and changes the header to indicate that the cell has been copied (shown by the appearance of an F in each header in Figure 3).

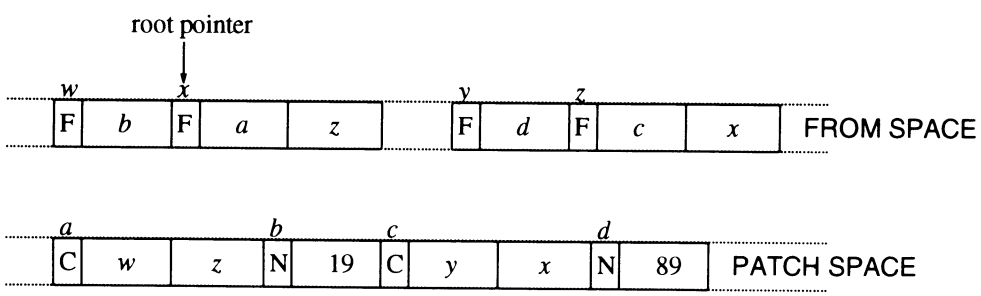

Figure 3.

The second phase sweeps through PATCH SPACE writing the cells to the output stream. All pointers are relocated to be offsets from the start of the stream by the relocate_out function. Note that as these pointers were not forwarded to point into PATCH SPACE by the first phase, relocate_out has to follow each one into FROM SPACE to pick up the corresponding forwarded address (see Figure 4).

The third phase finds and restores all forwarded cells in FROM SPACE by a single sweep through PATCH SPACE. It guarantees to find all the cells in FROM SPACE by considering all pointers in PATCH SPACE.

Discussion

Cost

The sole significant space cost of the output algorithm is that of PATCH SPACE, as there is no recursion to consume large quantities of stack space. Indeed, if the non-current semispace of a copying garbage collector is available for use as PATCH SPACE, then the effective space cost is nil. No doubt there will also be some hidden space cost associated with buffering of the output stream. The input algorithm needs no space overhead above a sufficiently large RECEIVE SPACE. A contiguous RECEIVE SPACE could be ensured by transferring the size of the graph first.

The time cost of the output algorithm is dominated by the three iterative sweeps over the graph, one in each phase. The cost of each of these is linear in the size of the graph; the entirety of FROM SPACE is never swept. The input algorithm visits each cell exactly once; its time cost is also linear in the size of the graph.

The number of bytes of data transferred is exactly the same as the number of bytes occupied by the pointer structure in primary memory.

Minor improvements and variations

Although the output algorithm has been explained as three separate phases, the first two phases could be combined into one: cells could be written to the output stream as soon as they are swept in PATCH SPACE by the first phase. The third phase must remain separate as all forwarding addresses are needed right up to the end of the second phase. Combining the first and second phases should give a small speed-up, but to have done so here would have complicated the presentation. Also, the possibility of transferring the size of the graph at the start of phase 2 would be precluded, as that is not known until the end of phase 1.

Various assertions could be checked at certain times to verify the correct behaviour of an implementation. For example, after the third phase there should be no cells in FROM SPACE still containing forwarding addresses.

The third phase needs to visit every cell in FROM SPACE to restore the attributes that were overwritten by forwarding addresses. As presented, the third phase achieves this by visiting every pointer in PATCH SPACE, effectively visiting some cells in

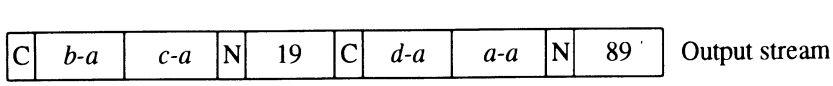

Figure 4.

FROM SPACE more than once. An alternative would be to have the copy routine push each cell in FROM SPACE, as it is first encountered, onto a stack. The stack links might be recorded by overwriting the cells' second attributes alongside forwarding addresses. This alternative would avoid any extra space cost, apart from a single top-of-stack pointer, but would assume that all cells are sufficiently large to hold two pointers.

Output to other destinations

The algorithm as described copies the pointer structure to an output stream. To retain the copy in memory would merely require replacement of the write in the PATCH-to-STREAM phase by a copy into a third space. However, more efficient algorithms are known for copying LISP-like pointer structures.[4,5] Another interesting possibility would be to replace that write by code to copy the cell back into PATCH SPACE over the original; this would invalidate the third phase, but the combination of first and second phases would then have exactly the same effect as a garbage collector! Although splitting any garbage collector into two phases would be unlikely to save time, the space savings in code, from the first phase being common to both garbage collector and transferer, might be attractive in a system with many different cell formats.

Interaction with garbage collection

The algorithm combines nicely with any garbage collector to achieve locality and minimize thrashing in a system with virtual memory, since both can copy cells in the same order (breadth-first). A copy of a pointer structure created by one will generally be found to be in a contiguous region by the other, and so it will be accessed sequentially. An exception is where part of the pointer structure is shared with another pointer structure, as that part might have been copied contiguously with the other structure. Intervening modifications also preclude such ideal locality.

Exchanging multiple graphs

Multiple graphs can be transferred by a sequence of transfers of single graphs. However, this fails to preserve sharing of any substructure between graphs. One solution is to build a structure combining the graphs into a single graph that can be transferred all at once and so retain sharing. This solution will not be appropriate if the graphs are not all available for transfer at the same time. Sharing of substructure can be preserved in such situations, but doing so requires some extensions to the algorithms as are presented below in the next major section.

Implementation as library routines

Garbage collection algorithms tend to assume a separation between the definition of new data types and their representations in cells. Only a few kinds of cells are needed to represent values of any data type. Since a garbage collector operates on the underlying cell representation of data, it can be written once as a library routine for inclusion in any program. If the same assumption is made for transfer algorithms,

then they too can be implemented once and provided as library routines. A single pair of transfer routines really can transfer any pointer structure.

Supporting a persistent store

Given that transfers can take place via secondary storage, an arbitrary time delay could be introduced between output of a pointer structure and its subsequent input. It seems that the binary transfer algorithms might be useful auxiliaries in the implementation of a persistent store. Further work is required to investigate this.

REFINING THE ALGORITHMS TO BE LAZY

Motivation

The new algorithms described above transfer entire pointer structures, i.e. the cells transferred are all those accessible from a particular root pointer. This behaviour is sometimes inappropriate. The particular examples we have encountered concern pointer structures used to represent programs and specifications. These pointer structures are usually called abstract syntax trees. A program or specification consists of a collection of definitions, and hence is represented by a collection of abstract syntax trees. Having been built by a parser, the abstract syntax trees are then traversed by a scope analyser to determine the bindings of references. Suppose that for each reference, the scope analyser sets a pointer attribute in the reference to point to the abstract syntax of the definition to which the reference is bound. Consequently the collection of trees become connected into a single graph. This is illustrated by the example definitions

```
x ≙ 42
y ≙ x
```

and the pointer structures in Figure 5 that represent them. In these pointer structures are three kinds of cell, distinguished by C for a cons cell, I for an identifier, and N for a number. The cons cells referred to directly by the root pointers can be thought of as representing the entire definitions. The third cons cell represents just the reference to x; it is a pair of the identifier x and the binding of that identifier.

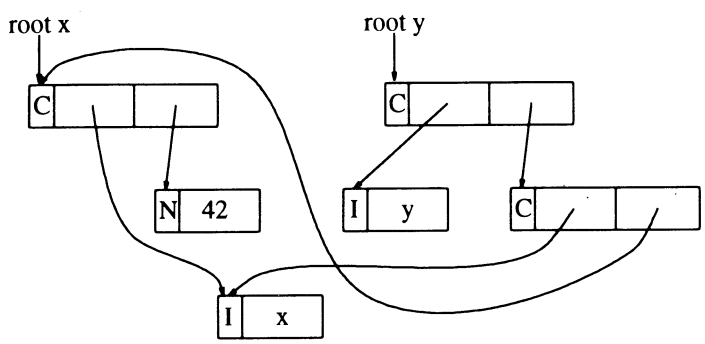

Figure 5.

Our intention was to transfer individual definitions, but a transfer of the definition of y would take that of x along with it. The pointers set by the scope analyser result in not just an intended definition being transferred, but all its auxiliary definitions too. Recursive definitions exhibit the same problem: the scope analyser creates cycles in the graph, causing the transfer of any one to include all the others.

Using reachability to decide what should be transferred is clearly insufficient in situations like the above. Somehow the pointer structure must be partitioned so that no more is transferred than desired. This is application dependent, and so some assistance from the programmer is required. In the example, the root cells of the trees representing definitions seem like good candidate places for partitioning the abstract syntax. Pointers representing bindings refer to these cells. The programmer characterizes them as those cells representing data of a particular type. However, assume that the transfer algorithms are unaware of the types—in the example, they are just cons cells—so how can partitioning be based on the types of data represented in cells? There has to be some distinguishing feature of the cells themselves (or of the pointers to those cells) for the transfer algorithms to recognise.

Suppose that having distinguished those places where transfers should cease, there remain different ‘partitions’ sharing a common substructure. In the example above, the identifier cell representing x would become a common substructure of the ‘partitions’ for the two definitions. (More generally, multiple definitions of the same name might share a common representation of that name, perhaps in a symbol table.) Should this common substructure have to be made into its own separate partition? Doing so would result in a greater number of smaller transfers, with the entailed extra process switching leading to loss of efficiency. Doing nothing would lead to the received partitions having separate copies of the previously shared structure. Could something else be done to retain the sharing without creating any more partitions?

After receiving one partition of a pointer structure, a process might find that it needs further partitions of that structure. A communication protocol must be established for these transfers. Must the originator anticipate the recipient’s requirements, or could the recipient request the needed partitions from a waiting originator? The latter possibility, transfer by need, can be characterized by the term *lazy*. In contrast, the previous transfer algorithms are *eager*.

The need for further partitions can be exemplified by block structured languages, where a definition can contain nested definitions. The nested definitions will each be in separate partitions from the rest of the containing definition. After the containing definition has been transferred, the recipient process might need some of the nested definitions, but the originator might not be able to predict which ones. The problem of designing a communication protocol for transfers of multiple partitions is addressed in the next major section; the remainder of this section is concerned with the problems of identifying and transferring individual partitions.

Outline of a solution

A solution that has been found to be satisfactory involves the introduction of two new distinguished kinds of cells, known as door cells and weld cells. Door cells are used to mark the entry points of partitions. Weld cells are used to mark shared substructure within partitions. For the earlier example, they would be used as shown in Figure 6. A D in a header denotes a door cell, and a W denotes a weld cell.

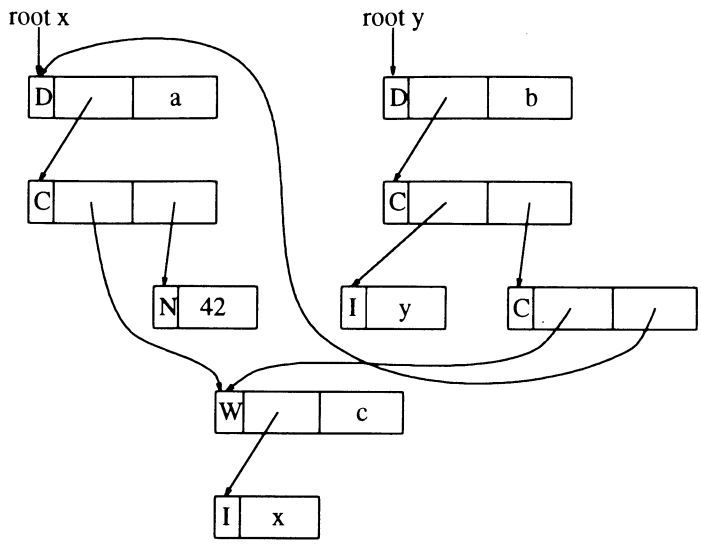

Figure 6.

Door cells behave like indirections: they contain a pointer to the root cell of a partition, which the user's code dereferences. This is the only pointer to the root cell of the partition; all other pointers that would have referenced that cell must now do so indirectly via the door cell. This pointer has a distinguishable NULL value if the partition has not been received yet. Weld cells similarly behave like indirections, but their pointers refer to an arbitrary shared substructure, not the root cells of partitions. Allocation of door and weld cells is the responsibility of user code. They tend to be needed on all values of only a few types, so the code to manipulate them can be cleanly hidden away inside the type definitions.

Each door or weld cell has a second attribute, which is a unique identifier that distinguishes the cell from all other door and weld cells. The unique identifier has two uses: to ensure sharing, and (in the case of door cells) to determine the process from which to request the partition that should lie beyond it. Sharing is ensured when a door or weld cell is received that has the same unique identifier as one already known in the receiving process. All pointers to the new cell are made to point to the first. This requires each process to maintain a mapping from unique identifiers to the first cell seen with that identifier. The second use of the unique identifiers requires them to be unique across all processes that allocate cells. It is eased by reserving a few of the most significant bits of a unique identifier to represent a so-called level number that identifies a process. As an example, we chose to reserve 6 out of 32 bits for the level number, hence imposing limits of 64 communicating processes and some 64 million door and weld cells allocated by any one process.

Weld cells can be viewed as exactly like door cells, except that the output algorithm always transfers the structure beyond them.

The pointer structure in Figure 7 shows what is received when the definition of y is transferred by these lazy transfer algorithms. If the definition of x is subsequently transferred, a pointer structure identical to that shown in Figure 6 would result.

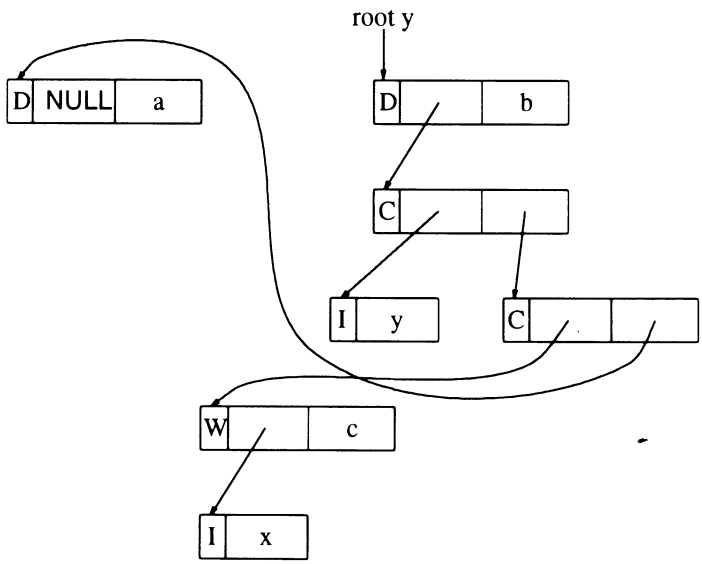

Figure 7.

The terminology of door cells and level numbers arises from an analogy between pointer structures and buildings. The partitions of a pointer structure are analogous to rooms in a building, with the door cells being the entry points (and exits) of rooms. The uniqueness of the identifiers allocated by different processes to door and weld cells is ensured by associating each process with distinct levels (storeys or floors) in the building—hence level numbers. Unfortunately, the terminology of weld cells does not fit the analogy.

Output algorithm

The new lazy output algorithm is largely similar to the earlier eager version. Indeed both might be found useful within a single application. Hence a parametrized output operation is presented that can perform a transfer either eagerly or lazily.

All three phases of the output algorithm treat door cells specially (weld cells need no special action). In lazy mode, phase 1 copies a single partition, including the partition's exit doors but excluding the partitions they reference. In eager mode, it has to beware of NULL-valued pointer attributes that can occur in door cells received from earlier lazy transfers. (Previously it was assumed that no pointer attribute could be NULL-valued.) Phase 2 generates the NULL pointers in lazily transferred exit doors. Phase 3 is as before but has to watch out for any NULL pointers. The auxiliary operations copy, relocate_out and refresh are exactly as before, and so are not shown again.

```
output(o: stream, r: from_pointer, mode:{EAGER,LAZY}) {
    initialize PATCH SPACE;
-- Phase 1, FROM-to-PATCH:
    copy(r);
    for each cell c in PATCH SPACE in order,
```

```
            for each pointer attribute p in c,
                unless mode is LAZY and c is a door cell,
                    if p is not NULL,
                        copy(p);
    -- Phase 2, PATCH-to-STREAM:
        for each cell c in PATCH SPACE in order,
            if c is a door cell and mode is LAZY,
                write c to o, but for its pointer
                attribute, substitute NULL,
                leaving c unchanged;
            else
                write c to o, but for each of its pointer
                attributes p, substitute relocate_out(p),
                leaving c unchanged;
    -- Phase 3, PATCH-to-FROM:
        refresh(r);
        for each cell c in PATCH SPACE in order,
            for each pointer attribute p in c,
                if p is not NULL,
                    refresh(p);
    }
```

Input algorithm

The input algorithm is responsible for maintaining sharing across partitions, both of door cells and weld cells. This is done in a sweep of the pointer structure after it has been received, and is aided by the mapping from unique identifiers to previously existing cells. Pointers to door and weld cells that have been seen before are relocated, causing the new copies of those cells immediately to become garbage. New door and weld cells are noted in the mapping. The auxiliary operation relocate_in is exactly as before, and so is not shown again.

```
input(i: stream) returns pointer {
    initialize RECEIVE SPACE;
    for each cell c in i,
        read c from i;
        for each pointer attribute p in c,
            if p is not NULL,
                p := relocate_in(p);
    for each cell c read into RECEIVE SPACE,
        for each pointer attribute p in c,
            if p is not NULL and deref(p) is a door or weld cell,
                if p is in the mapping,
                    p := value from mapping;
                else
                  add p to the mapping;
    return a pointer to the first cell read;
}
```

Discussion

Cost

The output algorithm has the same space cost as before, and although there are a few extra tests to evaluate, the number of sweeps is the same. The input algorithm has the added cost of a linear sweep over the received pointer structure, and the space cost of the mapping from unique identifiers to cells. The latter's exact cost depends on the implementation chosen, and on the proportion of cells that are door and weld cells. A programmer should avoid introducing any more door and weld cells than necessary, as extra ones both increase the size of the mapping and increase the number of process switches needed to transfer a pointer structure. A further small cost in lazy transfers is the transfer of duplicates of a door cell for each partition that references it.

Implementation as library routines

The introduction of door and weld cells allows the algorithms to be provided as library routines able to transfer any pointer structure on behalf of any program. This seems to be preferable to any slight advantage that might be gained by making the transfer algorithms aware of particular features of the specific pointer structures to be transferred.

Initiating transfers of needed partitions

The above algorithms show what happens when a transfer of a partition is initiated, but not how the subsequent transfer of a delayed partition is initiated. This need happen in only one place, namely in the operation that projects through a door cell. If the projection operation finds the pointer within the door cell to be NULL, a transfer is initiated. The projection operation can then return the desired result to the user's code, which proceeds as if the partition had been there all the time. It is not quite so straightforward for the process that supplies the partition, which must be ready to service the request. It might also want to initiate another transfer, but this must be synchronized with the receiver's requests, otherwise partitions might become interchanged and hence be misinterpreted. The initiation of transfers by the projection operation can happen transitively through a whole chain of processes, for instance a process A might make a request to process B, which in turn has to make a request to process C in order to satisfy process A's request. The next major section presents one way of achieving the necessary synchronization.

Implications for garbage collection

The mapping from unique identifiers to door and weld cells should be scanned by the garbage collector. Any entries in the mapping for cells that have become garbage can be removed. Note, however, that partitions which might yet be requested by another process must not be reclaimed. This can be arranged either by keeping root pointers to the pointer structures containing these partitions, or perhaps by use of a distributed garbage collector. Most distributed garbage collection algorithms use some sort of proxy or link cells to keep track of inter-address space references. The function of these has some overlap with that of door cells and so they might be merged in some implementations.

Incremental revision of partitions

After having transferred a partition to another process, the originating process is still free to mutate that partition. Suppose the other process must be informed of the changes. The input algorithm given above would immediately throw away the new version of the revised partition, redirecting any pointers to it to refer to the old version of the partition. This does not achieve the desired effect in this case. We have not needed to address this problem yet; it remains unsolved.

Degrees of laziness

Lazy transfers have been explained as transferring a single partition at a time, whereas eager transfers communicate all partitions of a pointer structure in a single transfer. Intermediate degrees of laziness can be achieved by allocating the ordinal numbers of door cells from several distinct ranges, those whose partitions are to be transferred along with the first partition being from a higher range than that first partition's door cell. Instead of suspending a transfer whenever a door cell is reached, transfer is suspended only when a door cell with an ordinal number from the same or lower range is reached. In this scheme, higher ranges typically need to be larger than lower ranges. As an example, we have five ranges, the first being numbers up to 2^{10}–1, the second from 2^{10} to 2^{14}–1, the third up to 2^{18}–1, the fourth up to 2^{22}–1, and the fifth up to 2^{26}–1.

USE OF LAZY BINARY TRANSFERS IN A RPC MECHANISM

A remote procedure call (RPC) is an invocation of a procedure in another process. It involves the transfer of argument values and results across an inter-process communication stream. The processes could be executing on different processors, but need not be.

RPCs provide a way of structuring communication between processes. Instead of transferring data structures at arbitrary times in arbitrary directions, arguments are transferred when procedures are called, and results are transferred when procedures return. Usually a difficulty with RPCs is how to transfer complex data structures—most implementations do not even attempt it. For example, the Amoeba RPC[6] supports only an unstructured data buffer argument (although a user application could use a mechanism, such as described in this paper, to code a data structure within the buffer). A somewhat richer example is the Sun RPC system[7] which supports basic data types such as integers and character strings, but not more complicated types. With the binary transfer mechanism, the transfer of complex data structures is no longer a problem.

RPCs assume call-by-value semantics—argument values are copied to the remote process. Call-by-reference does not work because arguments passed by reference cannot be dereferenced in a remote process. Where procedure arguments are modified by the remote process (say by a call to read a portion of a file) the semantics of RPC is usually call-by-copy/restore, which has subtly different semantics from call-by-reference. The eager binary transfer algorithms are obviously applicable to RPCs. Using lazy binary transfers, an RPC mechanism that approaches 'call-by-need' semantics is attained: only needed partitions are transferred, and a partition that was transferred in an earlier RPC will not be transferred again (unless a

garbage collector reclaimed it in the meantime). Call-by-need is generally shunned in connection with imperative languages, because of the possibility of argument values being modified between the time of the call and the time when need is first realized, so not surprisingly some care must be exercised in its use.

Motivation for remote procedure calls

Software engineering involves structuring solutions to complex problems. The aim in structuring a solution is to identify and separate distinct functionality, leaving the distinct functions communicating through clearly defined interfaces. There is a range of choices in how to implement these functions and interfaces.

At one extreme, the *uncoupled* approach, the functions are each implemented by an operating system process. The operating system both ensures proper separation, by not allowing one process to interfere with the state of another, and takes care of scheduling the processes, perhaps permitting some concurrency in execution. The possibility of distributing the processes over multiple processors is a further advantage. However, the interface mechanisms provided between processes are likely to be no more than untyped byte streams, requiring the processes to perform formatting and parsing of all data to be communicated.

At the other extreme, the *tightly coupled* approach, all the functions are implemented within a single operating system process. This allows arbitrary data structures to be communicated across interfaces. Such interfaces can be subject to strong type-checking, but despite this there is minimal protection between the functions. For example, different functions can share the same data in the same memory, hence changes to the data by one function may interfere with another. The lack of enforcement of proper separation makes it all too easy to do only a half-hearted job of the structuring. Moreover, after compilation and linking, there is no freedom for the end-user to compose functions in unanticipated ways, as there may be with uncoupled processes.

By exploiting the binary transfer algorithms, the disadvantage of the uncoupled approach can be avoided: arbitrary data structures can be efficiently communicated between processes without the user having to worry about formatting and parsing. This difference makes the approach worthy of a different title; the *loosely coupled* approach. Note that tight coupling still has the efficiency advantage (assuming a single processor): any explicit communication is bound to cost more than sharing data structures.

Whatever mode of coupling is used, the interfaces between functions should have the same specifications, though their implementations will differ. Procedure calls are particularly attractive for specifying interfaces, as they may leave open the possibility of different modes of coupling. Tight coupling results from direct implementation of procedure calls. Loose coupling or uncoupling results from use of RPCs. If an interface exhibits the same behaviour whichever mode of coupling is used, then it is said to have *ideal transparency*[8]. This can be achieved despite the different argument passing semantics of the different modes of coupling, but may require careful design and the adoption of a restricted programming style. The desire for ideal transparency independent of programming style has been described by Tanenbaum[9] as the ‘holy grail of transparency’. Although the search for this holy grail is laudable,

we have taken a pragmatic approach. Our use of binary transfers for RPC arguments requires the programmer to adopt a restricted programming style.

Implementation of remote procedure calls using binary transfers

Lazy binary transfers can be used in RPCs without any changes to the usual basic structure of the mechanism,[10] as illustrated in Figure 8. Caller and callee are the two communicating processes. The user code in the caller is shown remotely invoking a procedure, denoted by service( ), in the callee. The procedure denoted stub( ) has the same name and parameters as the service( ) procedure, and organizes the sending of arguments and receiving of any result, using the binary transfer routines output( ) and input( ). The callee might have several procedures that can be called remotely, so stub( ) must also prefix its output with something for the despatcher to distinguish which service( ) procedure is being requested.

Use of lazy binary transfers requires the stub and despatcher to behave slightly differently from usual. The stub procedure initially transmits only the first partitions of the arguments. It must then service requests from the callee for further partitions of the arguments. The caller must distinguish these requests from the communication of the result. This too could be transferred lazily, and so the callee must then service requests from the caller for further partitions of the result. The synchronization necessary in lazy binary transfers is thus achieved quite easily in the synchronized communication framework of RPCs.

Sometimes a RPC is required between the same pair of processes as are already executing a RPC in the opposite direction. Handling this involves the stub in the first caller taking on the role of despatcher for the second call. Further calls in the reverse direction again can be handled in the same way. If stubs did not have to play the role of despatchers, the implementation of stubs would be dependent on only the callee, and could easily be generated from a description of the interface of the service( ) procedure.

There is an interesting implication for garbage collection: care must be taken to prevent reclamation of the space occupied by partitions that another process might yet request. As was discussed earlier, an explicit communication to say that no further partitions will be requested can both allow a garbage collector to be called safely and allow further computation to be resumed.

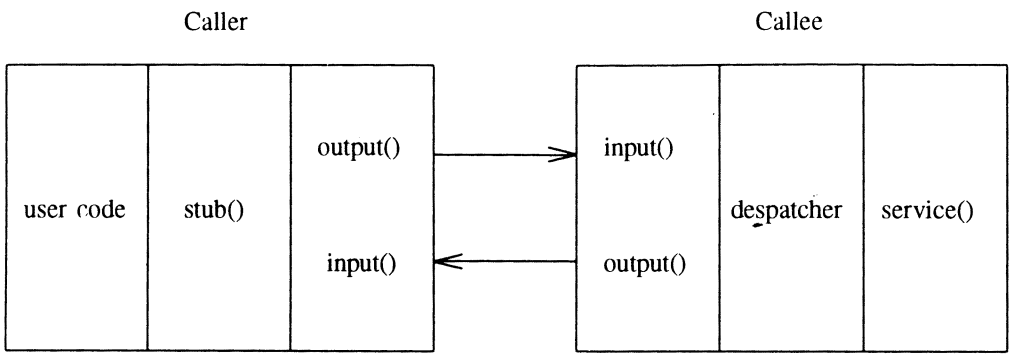

Figure 8.

Use of door cells and lazy transfers can result in differences from the call-by-value behaviour normally observed with RPCs and eager transfers. Firstly, a partition might be modified between initiation of a lazy transfer and the time when that partition is transferred. Secondly, a partition with a door cell already known to the receiving process is not transferred again, so any modifications to it in the meantime will not be made known. These possibly observed and possibly hidden modifications are difficult to exploit usefully.

How can avoidance of these modifications be guaranteed? Completing all modifications before initiating any transfers would be sufficient to provide such a guarantee, but is probably a stronger condition than necessary. Careful analysis is needed before allowing later modifications. If it is clear that a partition within a transferred structure will never be needed by the receiver, then modifications to that partition will be safe. It is also safe to modify a partition right up until initiation of the transfer of the first structure that contains it. The partition might not actually be transferred until a later RPC: modifications in the meantime would not necessarily be safe. Allowing further modifications after its transfer would be acceptable as long as it is never to be transferred as part of any structure to the same receiver in future. However, this time cannot be determined with confidence, as a receiver must receive the same data however long it delays requesting it, and it is very hard to determine how long that delay might be.

The above conditions go some way towards achieving an interface that has ideal transparency. For this to be achieved, neither the receiving or transmitting process may modify a value while the other might yet want to inspect the original transferred value. One programming style that satisfies this condition is for the transmitting process to complete all modifications before transmitting, and for the receiving process not to do any modifications. Both processes are then able to inspect the data for as long as they like. Another style that satisfies the condition is for the transmitting process to make no further use of the data after initiating its transfer, so that it will not be affected by any modifications made by the receiving process.

Use of remote procedure calls in a software engineering environment

This paper began by suggesting two very useful but infrequently provided tools: one for tracking the uses of declarations, and one for inspecting the types of expressions. To demonstrate the usefulness of the binary transfer algorithms embedded in a RPC mechanism, the paper now ends with a description of an existing set of tools that include instances of the suggested ones.

CADiZ is a set of tools that manipulate Z specifications.[11] As well as multiple parsers (e.g. yytroff and yytex) and formatters (e.g. swtroff and swtex), it has a type-checker (check), a user-interface manager (browse), and tools that expand schema calculus expressions (fullexp), simplify predicates (onepoint), and assist the proof of conjectures (theo). The architecture of the set of tools is illustrated in Figure 9. The arrows between processes denote the use of RPCs, which involve the transfer of abstract syntax trees.

The browse tool initiates the first RPC, and thereafter co-ordinates the activities of the other tools. It presents the user with a view of the parsed and type-checked Z specification, allowing pieces of the formal mathematics to be selected and inspected interactively. Tracking of declarations and inspection of types is supported

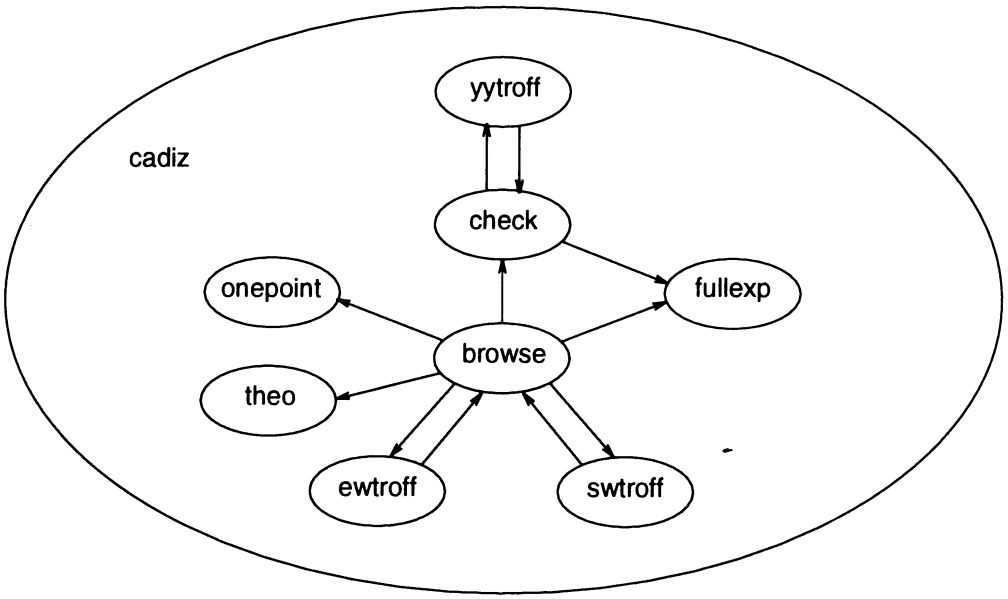

Figure 9.

by the browse tool, using information that the check tool necessarily computed and recorded in the abstract syntax trees. These are particularly simple operations for browse to offer. More complicated operations, such as expansion and simplification, are implemented by browse making RPCs to other tools.

The only extension to the type-checker, check, to provide the new functionality has been to make it ship out completed abstract syntax trees. In fact the type-checking tool is simpler in this architecture, because of the separation of the parser yytroff. This has been important in providing parsers for other concrete syntaxes of the same language: the common type-checker ensures that the same type system is obeyed by all the different concrete notations. Alternative parsers and formatters have been written by other people, which clearly demonstrates the utility of the approach. This was eased by the interfaces being specified by small collections of procedures. The abstract syntax trees produced by check are of use to other tools besides browse, for example style analysers.

The separation of functionality across multiple processes not only ensures the definition of clean interfaces between well-separated components, but also allows for components to be reused in other (perhaps unanticipated) ways, and for components to be replaced by alternatives, e.g. the parsers and formatters of CADiZ. The only disadvantage of this loosely coupled approach relative to the tightly coupled approach of a single process architecture is the relative inefficiency of communication between components. This disadvantage has been overcome in CADiZ by the use of procedure calls to define the interfaces, and by carefully designing the interfaces so that the procedures behave correctly regardless of whether communication of arguments and results is by copying or by sharing—the interfaces have ideal transparency. A functional style of programming that avoids modifications to data structures has helped to achieve this. The type-checker does not modify the information provided

by the parser, but it does use updating operations to fill in derived attributes in the data structures such as bindings and types that the parser was not itself able to initialize. Other tools, such as fullexp, compute their results as new data structures, leaving their arguments unchanged.

Tight coupling can be used not only for the complete toolset, but also for communicating subsets of tools. Each tightly coupled configuration needs a specific main program with code to create the process network to complete the linking.

Before creating this architecture, we foresaw that there might be a problem regarding different tools wanting to record different information in the abstract syntax trees. The binary transfer algorithms do not provide the option of leaving behind unwanted attributes, or making space for new ones. Consequently, the abstract syntax data types contain all the attributes needed by all of the tools. Addition of a new tool to the toolset might require addition of new attributes to some abstract syntax data types. The tools manipulate the abstract syntax data structures through an abstract interface that hides the representations of those types, so that when new attributes are added to the types, the new tools need only to be re-compiled, not modified.

PREVIOUS SOLUTIONS

The most common approach to transferring pointer structures is to generate a human-readable textual representation of the information contained in the pointer structure, and for the receiving process to parse this, re-generating the pointer structure. The intermediate textual representation of the information tends to be bulky, and expensive to generate and parse. Different code is written for transferring each different type of node in a pointer structure. The complexity and cost of this approach makes it little if any better—for the intended purpose of keeping extensions to a tool separate from the original tool—than duplicating the functionality of the existing tool.

Some alternative approaches have been developed in the context of the interface description language IDL. Lamb's approach[12] generates routines to do the output and input from the IDL type definitions. It encodes the data in a binary format for greater efficiency, yet retains the ability to perform transfers between different kinds of machines. Newcomer's approach[13] uses a single pair of output and input routines that operate on a tagged representation of data. It transfers data using the same binary representation that is used in primary memory, but with the pointers replaced by indices into a vector. The elements of the vector are the offsets into the stream of data being transferred of corresponding nodes in the pointer structure. The vector itself must then be transferred along with the pointer structure. This achieves greater efficiency than the other approaches, but loses the ability to perform transfers between different kinds of machine.

Herlihy and Liskov[14] use a binary transfer algorithm in communicating values of abstract data types around distributed systems, converting between different representations of data at different processors. It translates pointers directly to stream offsets on output, but requires a dictionary of all data in the input algorithm.

The approaches of Newcomer and of Herlihy and Liskov have similarities to ours. The differences are mainly due to the method used to relocate pointers. The other algorithms use an explicit dictionary for this, in one case having to transfer the dictionary between the processes. In contrast, the new algorithms make a copy of

the pointer structure, using the original as a perfect dictionary. Not only do the new algorithms avoid the transfer of a dictionary between the processes, but they also avoid the use of additional data types to represent the dictionary. Like Newcomer's, the new algorithms assume that the same binary representation is appropriate at both ends of a transfer.

We are not aware of any previous solutions to the lazy transfer problem.

CONCLUSIONS

Algorithms for the transfer of arbitrary, possibly cyclic, pointer structures between processes have been described. They are particularly attractive not only for their efficiency but also because they can be implemented by library routines able to transfer any pointer structure: no new code needs to be written to transfer values of a new data type. With some assistance from the programmer, transfers of large pointer structures can be restricted to smaller partitions, with further partitions transferred as and when needed. The transfer algorithms are an ideal auxiliary to a remote procedure call mechanism. This supports a loosely coupled process architecture, which is a particularly appropriate basis for software engineering environments, offering increased productivity, correctness and flexibility.

acknowledgements

Comments from Colin Runciman led to substantial improvements to the presentation in an earlier version of this paper. This version improved with comments from Jonathan Moffett, John McDermid and David Cattrall, and from anonymous referees. Financial support was provided by SERC SCHEMA Project (GR/G49531).

REFERENCES

1. R. R. Fenichel and J. C. Yochelson, 'A LISP garbage-collector for virtual-memory computer systems', *Communications of the ACM* **12**(11), 611–612 (1969).
2. P. R. Wilson, 'Uniprocessor garbage collection techniques', in Y. Bekkers and J. Cohen (eds), *International Workshop on Memory Management, Springer Verlag Lecture Notes in Computer Science 637*, September 1992, pp. 1–42.
3. C. J. Cheney, 'A nonrecursive list compacting algorithm', *Communications of the ACM*, **13**(11), 677–678 (1970).
4. D. A. Fisher, 'Copying cyclic list structures in linear time using bounded workspace', *Communications of the ACM*, **18**(5), 251–252 (1975).
5. D. W. Clark, 'A fast algorithm for copying list structures', *Communications of the ACM*, **21**(5), 351–357 (1978).
6. A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen and G. van Rossum, 'Experiences with the AMOEBA distributed operating system', *Communications of the ACM*, **33**, 46–63 (1990).
7. SUN Microsystems, *Network Programming Guide*, 1990.
8. S. Wilbur and B. Bacarisse, 'Building distributed systems with remote procedure call', *Software Engineering Journal*, **2**(5), 148–159 (1987).
9. Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall Int., 1992.
10. A. D. Birrell and B. J. Nelson, 'Implementing remote procedure calls', *ACM Transactions on Computer Systems*, **2**(1), 39–59 (1984).
11. D. T. Jordan, J. A. McDermid and I. Toyn, 'CADiZ—computer aided design in Z', *5th Oxford Z User Meeting*, Springer-Verlag Workshops in Computing, December 1990.
12. D. A. Lamb, 'Sharing intermediate representations: the interface description langauge', *CMU-CS-83-129*, Department of Computer Science, Carnegie-Mellon University, May 1983.
13. J. M. Newcomer, 'Efficient binary I/O of IDL objects', *ACM SIGPLAN Notices*, **22**(11), 35–43 (1987).

14. M. Herlihy and B. Liskov, ‘A value transmission method for abstract data types’, *ACM Transactions on Programming Languages and Systems*, **4**(4), 527–551 (1982).