CHANGE HISTORY

This is the first version.

TABLE OF CONTENTS

1. Purpose 1

2. Executive Summary 1

3. An Overview of Garbage Collector Technology 1
- 3.1 Fundamentals 1
- 3.2 Common Basic Techniques 2
  - 3.2.1 Mark and Sweep 2
  - 3.2.2 Mark and Compact 3
  - 3.2.3 Copying Garbage Collectors 4
- 3.3 Implementation Optimizations 4
  - 3.3.1 Conservative Collectors 4
  - 3.3.2 Incremental Collectors 5
  - 3.3.3 Generational Collectors 6

4. Responsibilities of TDF 6

5. Identifying Objects in TDF 7
- 5.1 Tagging 7
- 5.2 Object Headers 7

6. Supporting Garbage Collection in TDF 8
- 6.1 Conservative Garbage Collection 8
- 6.2 Relocating Garbage Collection 8
  - 6.2.1 Variables on the Stack 8
  - 6.2.2 Relocation of simple objects 9
  - 6.2.3 Expected Cost and Benefits 10
- 6.3 Generational or Incremental Garbage Collection 10

7. Implementation Development Strategy 10

8. Future considerations 11
- 8.1 Relocation of code 11
- 8.2 Relocation of stacks 12

9. Conclusions 12

10. References 12

1. Purpose

The purpose of this document is to describe possible techniques to support garbage collection with the prototype Dylan to ANDF producer. For each technique, the underlying TDF primitives are explained, and an evaluation is made of the expected efficiency of the technique compared to a corresponding native code implementation. The evaluation is limited to garbage collection in a strictly serial single threaded implementation.

The work was sponsored by the Commission of the European Communities.

2. Executive Summary

The programming language **Dylan** [Dyl92] requires the presence of a garbage collector. This is a requirement for many other languages also - but it is unique to Dylan amongst the languages considered in the GLUE project.

Garbage collection is normally implemented by a run-time system, which is linked together with compiled code written in the garbage collected language. Depending on the algorithm in use, there may also be a need for some support from the compiled code itself.

There are too many different garbage collection algorithms and techniques to discuss them all here. Instead, an overview is given of the main classes of algorithm, followed by a discussion of how such algorithms might be supported in TDF - both for the run-time system and for the compiled code. An evaluation is made of the expected efficiency of the implementation compared to a similar one using a native code compiler. Each of the algorithms can be implemented using a combination of core TDF support, architecture specific token definitions, and architecture specific libraries. TDF does not provide a fully architecture neutral solution to garbage collection - but this is probably not surprising to discover, since the best garbage collection algorithms tend to make extensive use of the API facilities of the target architecture, and TDF is not attempting to provide an architecture neutral API.

The expected efficiency of the TDF supported solution varies depending on the level of compiler interaction with the run-time system. For the least interacting algorithm, there should be no loss of efficiency. For the more complex algorithms there will probably be some loss of efficiency at function call time, and also a similar increase in code size.

3. An Overview of Garbage Collector Technology

3.1 Fundamentals

For many traditional languages, programmers are required to explicitly free heap allocated memory when it is no longer required. This approach leads to a simply expressed definition of *liveness* for an allocated object: an object is *live* until it is explicitly freed. Unfortunately, although it may be easy to express the liveness of an object in these terms, it can still be a very difficult problem for the programmer to determine the real lifetime of an object - and hence when to free it.

In contrast garbage collected languages make the freeing of heap objects automatic. The programmer does not have the responsibility for determining when the lifetime of an object ends - so the garbage collector must be able to determine this instead.

A garbage collector must determine which objects are live and which ones are garbage. This is known as *garbage detection*. The space occupied by the garbage objects will then be reclaimed (known as *space reclamation*), so that the program may use it again. Depending on the algorithm used, these two phases may be interleaved. The techniques used for space reclamation depend on the garbage detection technique. In many algorithms live objects are moved during garbage collection, so the reclaimed space does not necessarily correspond to any particular garbage object. In some algorithms, the space reclamation occurs implicitly as part of the garbage detection algorithm.

Garbage detection requires a criterion for determining the liveness of an object. The theoretical criterion would be that an object is garbage if it will never be used again by a running program. This might be impossible to determine in practice - even assuming an optimizing compiler which does control flow and data flow analysis. In practice, most garbage collector algorithms use a simpler, slightly pessimistic criterion, defined in terms of a *root set* of objects, and *reachability* from those objects. The root set represents all the conceivable starting points from which a program might find a path to an object. This typically includes global variables, and all local variables of active functions at the time garbage collection is invoked (typically, when an attempt is made to allocate an object, and there is insufficient free space). An object is reachable if it is either pointed to by a variable in the root set, or if there is a pointer to it from another reachable object. The process of following all the pointers of an object to find all other reachable objects is called *tracing*.

Determination of liveness by tracing all objects from a root set is a safe approximation, as the root set is guaranteed to include all those starting points from which the program **will** find a path. Similarly, the running program will actually only encounter a subset of all the reachable objects.

Evaluating the relative performance of different algorithms can be very difficult. The performance depends on details of the hardware and operating system being used - details of data caches and virtual memory paging methods can be particularly important. Performance will depend on the application itself too. For instance, if an application is designed to process data for an indefinite period of time, then it will probably be necessary to choose a garbage collection algorithm which compacts objects into a contiguous memory space, to avoid fragmentation problems. In contrast, an application which is short lived may get better performance with a simpler garbage collector which never moves objects. Another application may have particular real-time or interactive performance requirements, which require a form of incremental garbage collection. If an application is very short lived, then the best performance may be obtained by not having a garbage collector at all. This is why no single algorithm has emerged as a clear winner. This also means that it is important that TDF can support many different algorithms.

3.2 Common Basic Techniques

This section briefly describes the most well known garbage collector algorithms. For each general algorithm described, there may be many possible refinements or optimizations. The intention here is not to give an implementation guide - but rather to explain the basic approach. For a more complete study of garbage collector techniques, see [Wilson]. Section 6 shows how the important features of the algorithm may be supported by TDF.

3.2.1 Mark and Sweep

Mark and Sweep collectors are so named because of the way they implement the garbage detection and space reclamation, respectively.

Garbage detection is performed by scanning all objects (those in the root set, and all objects reachable from there). Each object encountered is then marked in some way - typically by setting a bit in the header of the object. Once this phase is complete, live objects will have been marked, while garbage objects will not.

Space reclamation is performed by sweeping the memory (i.e. scanning it exhaustively), looking at each object. Each object encountered in the sweep is then tested for liveness (by looking at the mark bit). If the object is live, then the mark bit is unset for next time. If the object is garbage, then the object is appended to a list of free objects (after possibly coalescing it first with any neighbours which are also garbage). Note the implicit assumption here that it is possible to find the start of every object during the sweep.

Allocation is performed by searching the list of free objects for a suitably sized block. This block is then unchained from the free list and initialized as a new object. If no such free object is available, then the garbage collector is invoked in the expectation that a suitable block will have been freed. If there is still no suitable block available, then the heap size must be extended with a suitable operating system call.

Mark and sweep collectors never move objects in memory, and hence are relatively simple to implement, and can give good performance. However the algorithm does have some drawbacks, which might rule them out for some applications. First, the fact that objects are never moved may lead to fragmentation problems, as discussed above. Secondly, the sweep phase has a cost which is proportional to the total number of objects (either live or garbage) in heap memory. Since, in most situations, there are more garbage objects than live objects this is potentially serious compared with an algorithm where the cost is proportional to the number of live objects. In practice, though, this may not be a problem since the sweep phase tends to be much faster than the mark phase.

A mark and sweep implementation requires a minimal amount of support from the compiler. The runtime system must be able to determine the root set, so normally the compiler must ensure that the runtime system is aware of all global variables and can find all the local variables for each stack frame. As will be seen in section 3.3.3, the runtime may be able to infer this information for itself without any help from the compiler.

3.2.2 Mark and Compact

The mark and compact algorithm is a modification of mark and sweep to stop it from suffering the fragmentation problems. The first phase of garbage detection is performed in the same way, by marking objects. However space reclamation is performed by compacting the live objects - for instance by shuffling downwards each live object encountered during the sweep, so that it is contiguous with all other live objects encountered so far. Once the compaction has been completed, the free space is also in a contiguous area immediately following the last object. This makes allocation simpler than with mark and sweep, because there is no need to search a free list for a block of an appropriate size.

This is an example of a *relocating* garbage collector, since objects are moved during the compact phase. It is therefore necessary to *fix up*, or update, any pointers the moved objects. One way to do this is to use an extra pass to compute the intended new location of objects before moving them, recording this address in a reserved location inside the object. Another extra pass then scans all roots and objects again, updating the pointers they contain to refer to the new location - obtained from the still unmoved object. Finally, the actual compaction is performed. Hence the cost of the compaction can be high. For this reason, mark and compact collectors often behave as mark and sweep collectors most of the time, and perform a compaction only when it appears that fragmentation is becoming a problem.

Mark and compact collectors require more accurate information about the root set than mark and sweep. In particular the runtime system must be able to determine which variables contain pointers and which do not, so that non-pointer variables are not fixed up by mistake. The compiler can arrange for the runtime system to find the location of global variables at the time the program is linked, for example, by arranging for the variables to all be arranged in a contiguous area of memory. The compiler must arrange for the runtime system to find local pointer variables by ensuring that the relevant information is associated with the function frame, and by ensuring that all function frames can be found. In addition, the compiler must allow for the fact that local pointer variables may change over function calls, if there is any chance that an allocation request might occur as a result of that call.

3.2.3 Copying Garbage Collectors

In the simplest example of a copying collector, the heap memory is split into two contiguous *semispaces*. While the program is executing, only one of the semispaces, the *current* semispace, will be used to hold all current objects, and to allocate new ones. When there is no longer any room to allocate a new object, the garbage collector is invoked. The garbage collector will then, in a single pass, copy all objects in the current semispace (known to the garbage collector as *fromspace*) to the other semispace (*tospace*).

This is done by iterating over all objects in the root set, and copying each object found into tospace. When a copy happens, the new location is saved in the old object as a *forwarding pointer.* Each pointer inside the copied object is then examined: it will still be a pointer to fromspace, and must be fixed up to point to the new version in tospace instead. If the fromspace copy has a forwarding pointer, then this is used for the new address. Alternatively, the object is itself copied and fixed up, and then the pointer is updated to the address of this new copy.

Once all the root set objects have been processed in this way, all the objects reside in a contiguous part of tospace, with a contiguous area of unallocated memory at the end of the space. Tospace is then made the current space, and the garbage collection is then finished. Allocation then occurs by simply incrementing the pointer into the unallocated part of the current space.

Like mark and compact collectors, copying collectors do not suffer from fragmentation, because they move objects into contiguous areas. An advantage of copying collectors over mark and sweep collectors is that they avoid the sweep phase, and hence the cost of the algorithm is proportional is the amount of live data. In fact, the algorithm can be made to be arbitrarily efficient by increasing the sizes of the semispaces. The cost for this is the permanent requirement for the two semispaces - which doubles the memory requirement of a simple copying collector compared with mark and sweep.

Interaction will be required between the compiler and runtime system to determine an accurate root set of pointers - exactly as for the mark and compact collector.

3.3 Implementation Optimizations

3.3.1 Conservative Collectors

Conservative garbage collectors are designed in such a way that they do not require any compiler support. This allows them to be used with existing compilers for languages which do not require garbage collection - such as C or C++. For instance it is possible to take the compiled binary files for an application written in C and link them with a library implementing a conservative garbage collector, and hence get the benefit of automatic memory management. The library must contain an appropriate implementation of

malloc, which interacts correctly with the garbage collector. The implementation of *free* will presumably be designed to do nothing - although it might put the object on a free list.

Since the conservative collector gets no information from the compiler, it cannot get a very accurate description of its root set. Typically, conservative collectors treat all global variables and all of the stack as roots. In addition, the conservative collector is given no information about where to find variables which contain pointers to other objects, and it must always allow for confusion with other values it might encounter (such as integers, or return addresses on the stack). The approach taken is to scan all the roots, and process any which look like they might be pointers (for instance, because they have a value which lies within the bounds of the allocation area). The assumption is that all processed objects might be live - even though it is possible that an object only appears to be live because an integer happened to look like a pointer. The algorithm is therefore conservative about what might be a pointer (hence the name).

It is important that conservative collectors never move objects because that would require fixing up anything looking like a pointer, which might, in turn, mean changing the value of an integer. For this reason, conservative collectors are often based on the mark and sweep algorithm, and suffer from fragmentation in some applications.

3.3.2 Incremental Collectors

All of the algorithms described in section 3.2 involve stopping the program at irregular intervals in order to perform garbage collection of all objects. The program continues when a complete garbage collection has been performed. These algorithms may be very efficient in terms of the percentage of program time lost to garbage collection - but they may have very poor interactive or real time performance because the garbage collection time can be large.

Many modern garbage collectors manage to do small amounts of garbage collection very frequently (e.g. with every allocation request). This allows the garbage collector to appear unobtrusive - or even to guarantee real time performance. Such collectors are called *incremental* because instead of carrying out a single atomic garbage collection they perform it in small increments.

Incremental algorithms are based on the algorithms in section 3.2, but with modifications which allow the program to continue work in the middle of a garbage collection cycle. To the garbage collector, the program appears as a coroutine which is capable of modifying the very memory that the collector is attempting to collect. For this reason, the program is referred to as the *mutator.* The collector must make sure that it maintains a consistent view of memory with the mutator as it traces through all reachable objects.

The types of inconsistencies, and the techniques for avoiding them are too complex to describe in detail here. A good overview may be found in [Wilson]. As a simplification, the mutator can cause inconsistencies when writing to memory (assigning to values in objects). Similarly, the mutator can notice inconsistencies created by the collector when reading from memory.

Incremental collectors normally cause the mutator to take special action for one of these situations (either reading or writing), and attempt to ensure by design that the other situation never causes problems. If the mutator takes special action on reading memory, this is called a *read barrier.* Alternatively, if the special action is taken on writing, this is called a *write barrier.* The read barrier approach is normally more expensive to implement (since reading occurs more often than writing in most programs). So most incremental garbage collectors use a write barrier - although read barriers have been used successfully on specialized hardware.

The implementation of a write barrier varies depending on the details of the collector. A common technique is for the write barrier code to save in a table the address of the location in memory which is about to be updated. The collector then has information about how to avoid the inconsistency.

The write barrier is normally performed by in-lined code, wherever a store into memory occurs. The compiler must arrange for this code to be generated.

3.3.3 Generational Collectors

The basic algorithms described in section 3.2 may be improved for real applications by making them generational. The optimization is based on the observation that in most applications, most objects live for only a short amount of time - although some objects live much longer. Most objects actually become garbage before a garbage collection occurs - but those which survive more than one collection are very likely to survive many collections. The intention behind the optimization is to avoid processing the longer lived objects during every collection because they are probably still live and there is little to be gained by attempting to collect them. This is done by segregating memory into two or more *generations*, and grouping objects in generations according to their age. Once an object has survived some number of collections in one generation it will be *promoted* to an older generation. The garbage collector will normally only attempt to collect the youngest generation, although it may decide to collect any number of consecutive generations including the youngest, to avoid garbage building up in other generations.

In order to collect the younger generation(s) without collecting the older one(s), the basic algorithms may be simply modified to ignore pointers to objects which are in too old a generation. (The generation may either be determined by storing it as an integer with the object itself, or, if all objects in a generation are in a unique memory space, by comparing the object's address with the memory bounds of the generation.) However, the liveness of the objects cannot be determined simply from the normal root set if older generations are ignored, because there may be a pointer from an older generation object to a younger generation one. Such pointers must also be considered as roots for the garbage collection, and so they must be recorded somewhere. This may be done by recording the possible location of such a pointer whenever an assignment occurs. The location may then be used as a root, and is subject to processing and fix up in the normal way.

Recording of pointers in this way is conceptually equivalent to using a write barrier, as described in 3.3.2 - and may be implemented in the same way.

4. Responsibilities of TDF

It has already been observed that garbage collection is normally implemented by means of a runtime system and a co-operating compiled program. If the final phase of compilation is actually a TDF installer, then this installer must be able to generate appropriate code to support the runtime. Hence it must be possible to express this support in TDF. TDF has the responsibility for expressing all the forms of support discussed in Section 3: indicating the locations of pointer variables for both local and global variables, and implementing write barriers.

For efficiency reasons, the details of the support may vary from one architecture to another. For instance the implementation of a write barrier might be managed by hardware alone on a machine with support for trapping writes to individual pages of memory. Such details may be implemented either in the installer itself (in which case core TDF expresses the required semantics), or by means of architecture specific token expansions. From the point of view of the compiler, it makes no difference which approach is taken. The interface between the compiler and the runtime system will be either a core TDF construct or a token

definition. The only important consideration is that it is possible to design such an interface which can work on any platform, provided there is a suitable installer and environment of token definitions. This means that the compiler must make assumptions about the availability of a such an environment for each target architecture, in the same way that it must assume the installer itself.

This document does not address the issues of who is responsible for ensuring that a suitable environment exists - although this issue is clearly of vital importance if the TDF approach is to be of benefit to end users of the technology wishing to port applications from one machine to another.

Similarly, this document does not address the issue of whose responsibility it is to provide the runtime system. It is possible to provide a runtime system which was compiled completely independently of TDF, and then link it together with the output from the TDF installer. The Dylan compiler simply has to assume that an appropriate runtime system will be present.

5. Identifying Objects in TDF

Like Lisp, Smalltalk, and some other languages, Dylan is a dynamically typed language. Dylan requires that some variables be allowed to hold values of any type. This means that it is not always possible for the compiler to determine whether a variable or memory location will be used to contain a pointer or an integer - or perhaps even both at different times. Implementations of Dylan must ensure that the type information is available at runtime. The garbage collector may require this information - as well as the type checking mechanism of the language implementation itself. See [G4.2.1] for a more detailed study of the requirements of a dynamically typed language.

5.1 Tagging

Dylan objects will normally be represented as pointers to heap allocated structures. These structures will contain a field which identifies the class (or distinguishing type) of the object. For efficiency reasons, it is important that some classes of object do not require heap memory for their representation. For instance, integers and characters are normally represented as *immediate* values, not as pointers. All Dylan objects are therefore represented as *tagged* values.

Some possible tagging schemes are described in [G4.2.1]. The details of the tagging scheme implementation are likely to be architecture dependent, so the support for the scheme will be provided by a tokenized interface which may have specific token expansions for different architectures.

The garbage collector runtime system may also have to determine whether a variable holds an immediate value or a pointer. It may either do this by using the same tokenized interface as the type checking mechanism, or alternatively with some specific code directly written in the high level language used for implementing the collector. All that matters is that the interface between the compiler and the runtime system is consistent, so that both agree about what is really a pointer.

5.2 Object Headers

Some objects contain fields which are not tagged Dylan values, and hence should not be traced by the collector. A string is a likely example of such an object - its fields would normally contain bytes. The garbage collector must therefore be able to determine which objects contain tagged values and which do not. This might be achieved by setting a bit in the header of the object. The runtime system will test this bit before deciding whether to trace any reachable objects.

6. Supporting Garbage Collection in TDF

6.1 Conservative Garbage Collection

It was seen in section 3.3.1 that conservative garbage collectors require no special compiler support, and hence require no support from TDF either. Since TDF is not contributing to the garbage collection mechanism, there should be no extra cost for providing this sort of garbage collector to a program compiled via TDF compared to a native code version.

6.2 Relocating Garbage Collection

It was shown in sections 3.2.2 and 3.2.3 that relocating garbage collectors must be able to determine an accurate root set, which must be exactly equal to the set of global and local pointer variables. The compiler must therefore output suitable TDF constructs or tokens to ensure that this is possible, and that such variables can be fixed up.

The compiler must also allow for the fact that any pointer value, whether in a local variable or in a field in an object, is subject to change across a function call.

6.2.1 Variables on the Stack

Harlequin's native implementations of garbage collected languages locate variables on the stack by context [G4.2.1]. The caller frame can be found by context from any function frame by following a chain of frame pointers. Similarly, the garbage collector traceable variables of a function within a frame are known to start at a fixed offset relative to the frame pointer, and are known to continue until a fixed offset from the next frame. Exceptions to this rule are also marked by context, for instance by using a uniquely tagged marker to identify a region of values which should not be traced.

The native implementation makes the assumption that values on the stack should be traced by the garbage collector by default, unless there is information to the contrary. This assumption is not valid within TDF, because the producer has no control over how the installer will arrange its temporary variables on the stack - and nor may it make assumptions about what values such temporaries might contain.

A more appropriate approach for supporting garbage collection within TDF is to assume that values on the stack should **not** be treated as roots by the garbage collector, unless there is information to say that they should. The producer must therefore arrange that each function provides information to the garbage collector to say what variables it has which contain tagged Dylan values.

Core TDF includes a mechanism to locate a local variable on the stack relative to the frame pointer (or *environment*) of the function invocation creating the variable. The TDF expression (**`env_offset`** *`variable`***`)`** gives the OFFSET of *`variable`* relative to the environment (this is a constant at install time), while **`(current_env)`** gives a POINTER representing the environment itself. It is a rule within TDF that if the offset of a variable is taken in this way, then the variable must be defined to have **`visible`** ACCESS.

In order to make this information available to the runtime system, the producer will enforce a calling convention in which the offsets of all the live, garbage collected variables of the caller are passed to the callee as an additional argument. These offsets are known as the *variable map* for the caller, and they are represented as a pointer to a constant structure which contains the number of variables, and an array of their offsets. As well as the variable map, the environment pointer of the caller will also be passed, as a second

additional argument. When the transition is ultimately made to use the TDF procedure extensions [G4.1.2], these additional arguments will correspond to *caller parameters*, and all Dylan functions will use caller parameters only for passing these additional arguments (normal Dylan parameters will correspond to callee parameters).

The following code fragments (in pseudo TNC notation) give an indication of how the map might be created, and also how the extra parameters are passed and received.

```
(make_var_tagdef function1_var_map
  (make_compound (shape_offset var_map_shape)
    num_vars_tok  (make_int ~signed_int 3)
    var_map_1_tok (make_int ~signed_int (env_offset function1_arg1))
    var_map_2_tok (make_int ~signed_int (env_offset function1_arg2))
    var_map_3_tok (make_int ~signed_int (env_offset function1_arg3)) | ))

(make_id_tagdef function1
  (make_proc (integer ~signed_int)
    var_map_ptr_shape - caller_var_map
    env_ptr_shape - caller_env_ptr
    object_shape  - function1_arg1
    object_shape  - function1_arg2
    object_shape  - function1_arg3
    |
    -
    (sequence
      ...
      (apply_proc (integer ~signed_int)(obtain_tag function2)
        (obtain_tag function1_var_map)
        (current_env)
        .....)
      ...)))
```

Since each function **F** is passed the environment pointer **P1** of its caller **C1**, **F** can also find the environment pointer **P2** for function **C2** (the caller of **C1**), since this is a parameter of **C1**, and may be found at a known offset from **P1**. By extension, **F** can also find the environment pointer of **C2**'s caller, and for all other functions in the calling stack. Similarly, **F** can find the variable map for each function. Provided that the garbage collector itself is called with this convention, the garbage collector can locate all variables on the stack.

This mechanism uses the calling convention itself to create a chain of saved environment pointers on the stack. This is more efficient than creating a chain where the head is a global variable (a mechanism which was discussed by Harlequin and DRA). But the mechanism does rely on the consistent use of the calling convention. There is a requirement for Dylan to be able to call, and be called by, other languages which do not respect this convention. This can be made to work by arranging that whenever a foreign call is made, the environment pointer chain is saved in a global variable. The convention can then be resumed by using this global value whenever a callback occurs into Dylan.

6.2.2 Relocation of simple objects

The garbage collector runtime system can locate all local variables on the stack, using the mechanism described above. Because all the variables are declared to be **`visible`**, the garbage collector may also modify these variables. This gives the garbage collector the same freedom to fix up pointers that would be

enjoyed by a native code implementation. The semantics of TDF ensure that the installer makes no assumptions about such values being preserved during a function call.

6.2.3 Expected Cost and Benefits

The convention described above for locating variables on the stack has an initial cost of two extra arguments to pass per function call. The importance of this cost will depend on the nature of the Dylan program itself. There is also a cost in terms of space, as the convention requires the variable location maps, which are constant arrays.

There may be an additional cost for the use of visible variables. This cost will depend on the installer used. On some platforms, this will make no difference. On other platforms some possible optimisations might be missed (for instance, storing variables in registers).

However, there is a possible benefit of this technique compared with Harlequin's native code techniques [G4.2.1]. It is possible for a function to be more accurate about liveness of variables at the time of making a call. The native code mechanism assumes that all tagged values still on the stack must be live. The map mechanism described above, however, allows a function to ignore those variables which it knows are no longer in use. This might allow the garbage collector to free more objects. It is impossible to quantify this benefit, as it will be program dependent.

6.3 Generational or Incremental Garbage Collection

It was shown in sections 3.3.2 and 3.3.3 that both generational and incremental garbage collectors normally use a write barrier, requiring code to be inserted by the compiler whenever a value is written into an object. For some specialist hardware it might be more appropriate to implement a read barrier instead - especially if the barrier can be completely implemented in hardware.

The implementation of these read or write barriers cannot be expressed in an architecture neutral manner. There is a strong dependency on the details of the garbage collector runtime system, and the hardware capabilities of the computer. However, it is possible to abstract the code generated by the compiler for reading and writing objects. A suitable target specific token definition can then cause the appropriate code for the read or write barrier to be installed.

It is expected that by using carefully designed token definitions in this way, it will be possible for TDF installers to generate comparable code to native compilers. Hence there should be no additional cost for supporting these types of collectors.

7. Implementation Development Strategy

Harlequin Ltd. are currently working on the implementation of a Dylan to TDF producer, as well as a design for a token interface between compiler and runtime system, encapsulating support for the most complex envisaged garbage collector algorithms.

It is an interesting observation that the power of the TDF token mechanism allows this interface to be completely abstracted, so that the details of the requirements for a particular runtime system can be satisfied by installing the TDF representing an application in the presence of an appropriate token definition library for that runtime system. Provided the interface description contains enough information, it may be used to communicate with any runtime system by inserting write barrier code or local variable maps as required.

Indeed, it is possible to use tokens to give an abstract interface to the layout of objects themselves - allowing runtime systems to use different object representations.

This form of late binding of compiler behaviour is not possible with traditional compilers. It is potentially extremely useful, allowing application writers to choose an appropriate garbage collector and object representation, and yet still make use of binary representations of standard libraries which also allocate or manipulate heap memory.

Harlequin will initially test this interface with a conservative garbage collector - with appropriate dummy token expansions for the compiler interface to the runtime system. The compiler can then be tested for implementation robustness without having to worry about garbage collector bugs.

An interface will then be written to a simple relocating runtime system, which will test the local variable mapping mechanism, and the accurate detection of pointers. Finally, an incremental collector will be tested.

8. Future considerations

The evaluation of garbage collector mechanisms discussed in this paper have so far assumed that the programs requiring support are complete applications. The program development phase introduces some extra complications which have not yet been considered. Use of the TDF technology for supporting programming development environments is outside of the scope of GLUE, so this chapter merely poses some questions for future study.

8.1 Relocation of code

A state of the art program development environment will allow the programmer to incrementally modify the code within a program as it is being debugged. Supporting this with TDF raises some interesting questions about how to load compiled code dynamically into a running program - but we will not address these questions here.

Assuming that a mechanism can be found for dynamically loading code, an appropriate mechanism should also be found for garbage collecting the old versions of code which have been modified. This introduces some new problems not encountered with the garbage collection of other objects.

If code can be garbage collected, then the root set must take into account any currently active functions - since these must also be considered to be live. The compiler must be able to express to the runtime system how to find the active functions. In native code implementations this is done by looking at the return addresses on the stack, which may be found at a known offset from each frame pointer [G4.2.1]. There is currently no mechanism in TDF to extract this return address.

Ideally, this garbage collector will support relocation of code objects - because a non-moving collector can cause fragmentation problems. This requires either that the code produced by the installer is position independent (not possible on some architectures), or alternatively that the code itself contains enough information for the garbage collector to relocate it.

8.2 Relocation of stacks

It can be useful to support the relocation of stacks - either in a development environment, or to permit the garbage collection of stacks in a language implementation of a parallel language. If this facility is available in a development environment, then a useful option for the programmer is the ability to switch to a new, bigger stack if a stack overflow is detected.

If a stack is relocated, then any pointers into the stack must be fixed up. In normal circumstances, the only such pointers would be in the stack itself (such as saved frame pointers), or in registers (such as the stack pointer or frame pointer).

TDF does not provide a mechanism for locating all such pointers, because the installer is free to implement stacks any way it chooses. However, there are standard ABIs for some architectures which might mean that the runtime system can infer this information for itself.

9. Conclusions

TDF is required to provide an interface between the compiler output and the runtime system, although the implementation of the runtime system itself is outside of the scope of TDF.

TDF is able to support all major forms of serial garbage collection, using a combination of core features and target specific token libraries.

It should be possible to mix TDF code produced by two different language compilers - one which supports garbage collection and one which does not - without compromising the efficiency of either language. This will require a suitable foreign language interface between the two languages.

There is a cost for using TDF as opposed to native code to support relocating garbage collectors. This cost appears in terms of speed, code and size. The cost comes from passing two extra parameters with each function call. One of the parameters will require the reservation and initialization of some static data. There is also an additional cost as Dylan variables will normally have to be declared to be visible. This will remove the opportunity to locate them in registers.

10. References

[Dyl92] Apple Computer, *Dylan - an object oriented dynamic language*, April 1992

[G4.1.2] Ian Currie, *Proposed Extensions to TDF for Ada and Lisp*, GLUE deliverable 4.1.2, DRA, September 1993

[G4.2.1] Tony Mann, *TDF support required by Lisp*, GLUE deliverable 4.2.1, Harlequin Ltd., April 1993

[TDFspec] DRA Malvern, *TDF Specification, Issue 2.1*, June 1993

[Wilson] Paul R. Wilson, *Uniprocessor Garbage Collection Techniques*, University of Texas, 1992