ESPRIT PROJECT 6062

Report on inter-language working in ANDF

Written by:	Tony Mann, Simon Green Harlequin Ltd.	________________________
Issued by:	Jorgen Bundgaard DDC International	________________________
Delivered by:	Gianluigi Castelli Project Director	________________________

The status of this document is "WORKING" if signed only by its author(s); it is "ISSUED" if signed by the Workpackage Manager; it is "DELIVERED" if signed by the Project Director.

Project deliverable id:	TR4.2.3-01
Document code:	Harlequin GLUE4.2.3 - 01
Date of first issue:	1995-01-24
Date of last issue:	1995-01-24
Availability:	Confidential

omi/glue

CHANGE HISTORY

This is the first version.

1. Purpose

The software development process is evolving due to ever increasing complexity and commercial pressures. One common trend is to develop software as reusable components, which are designed to inter-operate with other components. Dylan [DIRM94], a dynamic, object oriented language, is specifically designed with this trend in mind.

Dylan, like other high level languages, provides abstractions of programming concepts which must be made explicit in the more traditional languages at which ANDF is primarily aimed (such as C). The compilation of some of these abstractions depends upon the consistent use of conventions which must be carefully chosen by the implementor for efficiency. Features which require such conventions for their support include automatic memory management (garbage collection), exception handling and dynamic functions.

Components written in other languages will be unaware of any Dylan-specific conventions, so a Dylan implementation which supports mixed-language component integration must be tolerant of unco-operative behaviour across component boundaries.

Harlequin are in the process of developing a Dylan to ANDF compiler under task 4.2 of the OMI/GLUE project. ANDF is specifically designed as a format which can represent the partially compiled output of many different languages, such that the compilation process can be completed on many different platforms. However, in order to compile some of Dylan's conventions efficiently, we have found it necessary to extend ANDF.

The algorithms and ANDF extensions used by the producer to implement the Dylan-specific conventions were described in [G4.2.2a],[G4.2.2b], but the analysis was limited to homogeneous Dylan applications. In this document we identify the problems that may be encountered when a component written in Dylan is combined via ANDF with a component written in another language. We look at solutions to those problems, and offer guidelines for language implementation in general when multiple-language components must be supported. Finally, we examine the cost of those solutions and guidelines when implemented in ANDF.

A serial execution model is assumed throughout the document. Extensions to these techniques to support multiple threads are being examined in task 5.7.

This work was partially funded by the Commission of the European Communities.

2. Executive Summary

Harlequin's Dylan producer uses implementation techniques which are not used by producers for the other languages under development in the GLUE project. These techniques are designed to be used consistently by the producer to provide services which are not available with more traditional languages. Garbage collection, for example, is a Dylan service which requires a global understanding of a running application in order to determine the liveness of data and collect unreferenced objects.

Dylan is unlikely to be used as the only language for building any application, however. It is therefore vital for the commercial success of Harlequin's Dylan compiler that it is able to support inter-operability with other languages.

If Dylan producer output is naively linked with ANDF from another source, then the producer's expectation of consistency will not be met. Garbage collection might be expected to fail, for example, as the liveness of objects is a global property, and yet only functions created by the Dylan producer will be co-operating with the garbage collector. Even function calls might be expected to fail, as our Dylan implementation depends upon consistent use of an argument passing convention which is not used by other producers.

In this document, we show that the techniques we have developed to support homogeneous Dylan applications can be readily extended to support heterogeneous, mixed-language applications, provided that some care is taken by the Dylan producer when generating code which will interface with "foreign" parts. We present guidelines for inter-operability between mixed-language components in general. The guidelines restrict the types of conventions that complex components may depend upon, so that inter-operation with other components which do not follow the conventions is still possible. The guidelines impose minimal runtime cost, except at the interface boundaries of complex components.

We describe the details of the interface that makes inter-operation possible. The implementation of this interface is independent of the output format of the compiler, and hence can be described at a higher level than ANDF. We choose to describe it in terms of primitive extensions to Dylan, and we show how these primitives are mapped onto ANDF. Since ANDF is able to implement this interface in full, we conclude that *ANDF presents no obstacles to inter-operability.* Indeed, because of it's architecture neutrality, ANDF is a significant aid to inter-operability.

Not all inter-language interfacing problems can be solved with this model, however. There are some complexities related to exception handling and unusual control flow which have no obvious solution - either in ANDF or in native code. It is hoped that by documenting the existence of these problems they will not prove to be a significant obstacle to inter-operability.

We have tested calling-out across simple interfaces from Dylan to C using the described techniques with a low-level interface description. These tests constituted part of Harlequin's demonstration at the OMI conference in Dublin (7 - 9 November 1994). The work to support high-level interface descriptions and calling-in from C to Dylan is still ongoing, as is the Dylan standardization process on which it depends. The implementation techniques required to support this work in ANDF are now in place, and the remaining work is mostly limited to implementing macro support for the Dylan extensions to support the high-level interface.

3. How to Mix Components in Multiple Languages

3.1 What is a Software Component?

In the most general sense, a *component* is a replaceable, reusable unit of software. A library is an example of a component; it provides functionality which can be used in many contexts, and it may be substituted for another version. Components must have a well-defined interface, and in a good component-based design, each of the components will offer behaviour which is orthogonal to the others in the set.

Components are bound together by some means. In the case of a library, the binding is performed by a linker, to form a single executable or process. Components can be bound together by other means. Unix utilities interconnected via pipes, OLE parts interconnected graphically under Windows and network services via RPC are all examples of components with their binding mechanisms.

For the purposes of this document, we particularly focus on ANDF *capsules* as components, where the binding is performed by the ANDF linker. The discussion can be readily generalized to other types of bindings, though. The guidelines are appropriate for all component models, although the details of the interfaces between components may have to be modified for other architectures.

3.2 Inter-language Semantics

Since components have interfaces which are well *defined* in terms of some interface language, they may actually be implemented in any programming language provided that the chosen language can *specify* the interface in a way which is compatible with the definition, and provided that the compiler can *implement* the interface compatibly with the interface rules. In other words, compatibility is required at both the API and ABI levels, respectively.

Usually, ordinary programming languages are used as interface languages, and the interface rules are defined by either the ABI of a particular operating system, or the implementation of the programming language itself. For many general-purpose operating systems, it is common for the ABI to be defined in terms of the C compiler output, since operating systems are commonly implemented in C. For such a system, C is also typically used as the API. It is clearly possible to implement conforming components in C, since the interface definition and specification languages are the same, and the interface rules are defined in terms of the compiler's implementation.

Some component interfaces will be specifiable only in the programming language which defines the interface. For example, Dylan permits programmers to design components which can have their functionality extended by other components by means of high-level Dylan facilities such as inheritance and generic function specialization. It is unlikely that the Dylan API will be specifiable in any other language, so such components may only be written in Dylan. Components will very probably also require the consistent use of a single Dylan compiler for their correct behaviour because Dylan does not specify how these facilities are to be implemented in a portable manner. Interfaces which are specified in this way are thus not suitable for mixed-language binding.

Component interfaces which do not contain high-level language-specific features may potentially be implementable in many languages. If such an interface is defined in a suitable *neutral* API and ABI, then the component may be generated by any compiler which is able to specify the API and conform to the ABI. For the standard general-purpose operating systems, the *de facto* neutral interface is that used by the C compiler provided by the operating system vendor. ANDF itself also describes a neutral interface, but it is a very low-level language for defining an interface and hence may not be suitable for API definition. We assume here that the neutral interface of relevance is the *de facto* C-centric one, and note that ANDF installers are designed to be ABI compatible with this.

If we use C as the neutral interface language, then the semantics of inter-component communication are defined in terms of C. This implies that the semantics of inter-*language* communication are also defined in terms of C. This is simply a pragmatic choice, which allows us to concentrate on a single *foreign language interface (FLI),* in which the C interface definition is specified in Dylan.

Since ANDF is language neutral, at first sight it seems limiting to concentrate on C. However, we are able to interface to components produced by any compiler which also has a C interface. Given sufficient effort, it should be possible to interface Dylan directly to other languages, such as C++ or FORTRAN. This would permit the interface to specify any high-level language features which are present in both Dylan and the other language, but not present in C. This ability might allow a mapping between C++ classes and Dylan classes, for instance.

There are interface languages which are designed specifically to support the high-level abstractions of object-oriented languages in a language independent manner. IDL (*Interface Definition Language*) is one example. These languages are not discussed here in any detail — but it worth noting that they typically work by defining conventions for using the object oriented features in terms of APIs which are ultimately expressible in C.

The examples in this document refer to a C interface, reflecting Harlequin's current FLI effort. However the analysis does not depend on the details of the interface, and may be readily extended to other languages (such as IDL) and other component binding techniques (such as RPC), provided that the guidelines given below are not violated. An extension to another language in this way may require Dylan's FLI implementation to have knowledge of any specific ANDF tokens used by the foreign language producer.

3.3 Potential Problems

We have seen that the pragmatic and semantic problems of interfacing two languages may be solved by choosing a neutral API and ABI for the interface. However, there may still be problems of inter-operability due to higher level considerations. These fall into three broad categories:

3.3.1 API Limitations

There may be a mismatch between the semantics of the neutral language and the implementation language, such that the neutral API is unable to express a desired language feature. High-level languages, like Dylan, are particularly likely to suffer from this problem. This will limit the complexity of the component interface to the lowest common denominator of the features of the implementation and neutral languages.

In practice, the interfacing logic of the high-level component may be able to map between language features which are similar but not identical, but this may cause additional complexity or inefficiency. For example, if a C interface function returns a string to a Dylan component, it is probably important for the interface logic to map the C string (a pointer to a null-terminated byte-array) into a Dylan string (a self-identifying object with an object slot containing the string's size as well as the characters). The mapping will require the allocation of the Dylan object, and may require copying of the characters. This is particularly inefficient if the component on the other side of the interface also happens to be implemented in Dylan and hence has to do its own mapping in the reverse direction.

In general, when interfacing Dylan to C, the interface cannot support many of the dynamic and object-oriented features of the language. It is always likely to be more efficient and convenient to interface two Dylan components together directly, rather than via the C API.

3.3.2 ABI Limitations

Simple differences in the ABI are easily resolved by suitable interface logic. For example, if a Dylan implementation internally uses a calling convention in which arguments are passed as callee parameters [G4.1.2], then Dylan code may be interfaced by using stub functions which use callee parameters on the Dylan side of the interface and caller parameters on the C side.

Where a high-level language uses an extended ABI for supporting language features, ABI differences may be more complex - or even impossible to solve. For example, Harlequin's Dylan ABI is designed to support Dylan's dynamic features, including garbage collection, closures and multiple values [G4.2.1]. Some of these features cannot be used across the interface anyway because the API does not permit them either (multiple values, for example). Certain uses of Dylan's dynamic environment must be disallowed across the interface specifically to circumvent ABI problems — for example a Dylan function must not call a C function in the context an `unwind-protect` body if there is any possibility that the C function might perform a `longjmp` to transfer control to a point outside of the context of the `unwind-protect`.

The fact that C is statically linked imposes a more subtle ABI restriction. For compatibility, any Dylan interface implementation must also be statically linkable, hence a Dylan function which may be called across the C interface must be describable statically, ruling out the possibility of implementing it as a closure. If it were possible to create a closure statically, then the ABI would not, actually, prohibit the closure being used in this way — but Dylan has no static closure facility.

Garbage collection can be made to work in the presence of an ABI which offers no direct support (see Section 4.4) — but there are restrictions on the design of possible garbage collectors, and there may be a cost associated with the lack of direct support on the remote side of the interface.

3.3.3 Component Behaviour Assumptions

If an interface is genuinely both API and ABI compatible with the neutral definition, there still may be problems with the design of the component if it makes assumptions about the behaviour of other components which are invalid.

Consider, for example, a hypothetical memory allocator which is provided as a component called MA1. It requests memory from the operating system in large blocks, and allocates objects from the large blocks. The operating system guarantees that each successive block requested will be contiguous in the address space with the last, and the allocator is implemented with the implicit assumption that all blocks it requests will be contiguous.

Component MA1 may perform correctly in some circumstances, but as soon as it is used in a system with another component which also requests blocks from the operating system, its assumptions about contiguous memory will be invalidated. If an alternative implementation of the memory allocator, called MA2, is designed to be tolerant of gaps in its memory space then it is more likely to perform correctly when used in a system of components.

Note, however, that adding MA2 to a previously working system may still cause the system to break. For instance, the system may have been working correctly with MA1, and the addition of MA2 would then break the system by violating the assumptions of MA1. In a complex system, it may be impossible to isolate the blame for a failure to a single component, but the observation is that "good" components avoid making assumptions about what other components will do wherever possible.

Both language implementors and component programmers must be aware of the problem of making incorrect assumptions about the behaviour of other components, and must adjust their designs accordingly. This is in contrast to the API and ABI problems which must be solved by language implementors, and appear to component programmers as restrictions only.

3.4 Guidelines and Constraints

It is possible to build systems from reusable components without suffering from the integration problems described above. Inevitably, some constraints must be imposed on both the language implementor for a high-level language and the component designer. If all components are designed with these constraints in mind, system integration of components should be significantly less problematic. Unfortunately, it is particularly likely that older components will have been designed without following these guidelines.

It is outside the scope of this document to consider all component integration problems, as we are considering language implementation issues. But we note that the language implementor of a high-level language may need to design a run-time system component too, so it is useful to pay some attention to component design guidelines.

3.4.1 Language Implementor Guidelines

The language implementor is responsible for ensuring that components obey the rules of the neutral language at the interfaces to foreign components. Specifically function calls and data representations must be compatible with the neutral ABI, and must follow the definition given by the API.

There are two implementation choices for meeting this responsibility. Either languages will be implemented to be compatible with the neutral ABI at all times, or the implementor must arrange for a protocol conversion in the FLI. Implementors have the freedom to use any language-specific ABI within the code of a component, provided that the use of any language-specific conventions is not visible from the other side of the neutral interface. Note that implementors are permitted to use the language-specific ABI to interface components written in the same high-level language — the neutral interface is only necessary for foreign components.

The high-level features of Dylan require implementors to use a more complex ABI than C — so Dylan components will inevitably need a FLI to interface with C components and to hide Dylan's "unusual" implementation choices.

3.4.2 Component Designer Guidelines

Component designers must be aware that there may be other components present in the system about which they have no knowledge. It is valid to assume that these unknown components will, themselves, be designed with the component guidelines in mind, but it is dangerous to make other assumptions about them.

A helpful hint to component designers is to imagine another unknown component in the system performing identical tasks with an identical algorithm. If the pair of components are making invalid assumptions, then they are most likely to break when combined in this way (because components offering the same services are most likely to compete for the same resources).

The component designer is responsible for documenting the interface provided to clients of the interface. This documentation must include any restrictions in the use of the interface. Similarly, the component implementor must ensure that no restrictions are violated when interfacing as a client to other components or to the operating system. In particular, implementors must assume that other components might also be clients of the same services unless the documentation of the interface says otherwise. In general, it should be assumed that operating system services and resources may be used by any client.

3.4.3 Systems with Multiple Garbage Collectors

A garbage collector (GC) is always associated with a memory allocator, which will be provided as a service by a GC component. Typically, the allocator will make use of operating system services to claim memory for its purposes. If such an allocator creates an object in a system, then the associated GC will *manage* that object — it is responsible for collecting that object.

Any GC component must be able to *trace* all of the objects it is managing which are still *live* and collect all the non-live ones [G4.2.2a]. In order to do this it must document how it expects its clients to use allocated memory. This may include rules about the compiler support required from clients (in which case, it may not be permissible for clients to interface to the GC using a neutral interface alone). There may also be rules about how allocated memory may be passed across interfaces to other components which are not themselves clients of the GC.

For multiple GCs to co-exist in a system, it is necessary for each GC to be able to trace its own objects, without violating the rules of any other collector in the process (such as modifying the contents of an object created by another collector). It is assumed that it is always possible for a collector to read the contents of an object created by another collector — but that it is not valid to make any assumptions about what the contents mean. This places some restrictions on the types of tracing and collection algorithms which may be used.

Tracing algorithms for GCs work by starting from a set of *roots*, processing each object found there to mark it live, and following all direct references from the object to other objects, which are then processed in turn. However, a GC is only permitted to process an object it is managing. Each GC must, therefore, either arrange to detect and ignore objects it is not managing while tracing, or alternatively it must ensure that it will never encounter objects it is not managing while tracing (by documenting appropriate rules to disallow inappropriate pointers from being written there in the first place). Typically, a *conservative* collector [G4.2.2a] would follow the former approach, while a *total* collector would use the latter.

Conservative collectors can be used successfully in a multi-GC system provided that the algorithm is able to ignore objects which are not being managed, but continue to process them as possible roots. Total collectors can also be used successfully, but they may be forced to impose very restrictive rules to their clients, depending on the details of the algorithm used. For some collectors, it might be illegal to pass managed objects across a neutral interface, for example. The implications to garbage collectors of foreign language interfaces are discussed in more detail in section 4.4.

3.4.4 Mixed Systems with a Single Garbage Collector

In some carefully designed systems, it may be possible to combine components written in more that one language with a single garbage collector. Since, in general, a garbage collector algorithm depends on consistent co-operation from compiled code, there is a minimum rule that the compilers for each of the components requiring garbage collection must all co-operate appropriately according to the garbage collector’s rules, and must use the same API to invoke the garbage collector and its associated allocator.

If a mixed-language system is built with a single garbage collector in this way, then there is no need for any restrictions at the interfaces specifically to support garbage collection, provided that the normal garbage collector co-operation is maintained for cross-interface calls. This is an example of an interface language which is at a higher level than the neutral language (because it supports the co-operation), even though it may be definable and implementable in the neutral language.

In practice, most languages which require garbage collection have traditionally been implemented with custom garbage collectors which are designed to work efficiently with the language’s features. In all of Harlequin’s experience of dynamic language implementation, we have not yet encountered two language implementations with garbage collector interfaces which are similar enough to be implemented with a single collector component. (In fact, the need for this diversity is the main reason why ANDF does not attempt to provide its own direct support for garbage collection).

There is one case in particular which might be particularly suitable for implementation in this way, however. It is possible to implement a conservative garbage collector which requires virtually no co-operation from compiled code. Typically, there is just a minimum requirement which restricts the use of *derived* pointers (that is, pointers which do not point directly at the object itself) [BW88]. For many standard static languages, including C, this requirement will only be violated by the most optimizing of compilers.

In principle a single conservative collector can be used successfully in a multi-language system provided that each compiler meets the requirement concerning derived pointers, and provided that all the client components agree on the API. If it is possible to replace the standard implementations of C's `malloc` and `free` in a system, then these functions could be used as the API to the garbage collector (possibly with a null implementation for `free`). It may be possible to use garbage collection in this way even if the components were originally designed for manual memory management. We have successfully tested systems in this way with components written in Dylan and C, and using Boehm's conservative collector.

4. Significant Features of Dylan

The unusual features of Dylan were described in detail in [G4.2.1]. There are some features which are worth discussing again in terms of their impact on inter-operability, and what must be done to support them across a C interface.

4.1 Dynamic Typing

All Dylan values are self-identifying at the language level. This means that the implementation must store the type of the value alongside the value, either as tag bits or as an extra field in a data-structure, unless the type can be inferred from the context. For example, a double-precision floating point number in Harlequin's Dylan is represented as a pointer to a region of memory containing a reference to the class `<double-float>` as well as the 64 bits of floating point data itself.

For several common numeric types in Dylan, there is an equivalent type in C. For example, a `<double-float>` corresponds to the equivalent C type `double`. In these cases, it is possible to convert between the Dylan representation and the C representation by either adding or removing the type information, depending on the direction of the conversion. Values represented as dynamically-typed Dylan quantities are said to be *wrapped*, while values represented as C quantities are said to be *raw*. The Dylan interface code must be able to convert between raw values and wrapped values automatically, as appropriate.

For other types, a more complex coercion may be needed. The Dylan FLI (described in more detail in chapter 6) has built-in converters for simple flat types like strings. These conversions typically require allocating memory for the alternative representation and copying data from one representation to the other. The FLI also permits programmers to encode their own converters in terms of the simpler ones provided. By default, if no conversion is specified, the Dylan value is passed directly through the interface. In this case, the value must be treated as an opaque type in the foreign component; operations on the value may only be performed from within Dylan, so the value must first be passed back into a Dylan component.

4.2 Data Abstraction

Objects in Dylan may contain *slots*, which are analogous to fields in C's *structs*. Accessing these fields is done through function calls — although implementations can in-line the function in most circumstances, so the language does not imply that slot access will be inefficient. The details of the arrangement of these slots in memory is left to the Dylan implementation, and is encapsulated by the accessor functions themselves. There is no way in the C interface to describe how to access the slot, other than by passing the accessor functions themselves. Since the accessor functions are on the Dylan side of the interface, calls to these functions are permitted from C without violating the opaque type rules.

4.3 First Class Functions

Control is always passed between C and Dylan across an interface boundary by means of function calls and returns. But Dylan functions have properties which are not found in C functions, so some restrictions must be made about what types of functions can be mapped across the interface.

We refer to Dylan calling C as a *call-out*, and C calling Dylan as a *call-in*. A call-out function will be implemented in C (or compatibly with C) — and the FLI provides a stub interface function in Dylan which looks like a Dylan function and which calls the C function. Similarly, a call-in function is implemented in Dylan, and the FLI provides a stub function which looks like a C function and which calls the Dylan function.

One feature of Dylan functions is that they may be *closures* — that is, they may require an environment to represent the values or variables they have closed over. A call-out function will appear as a first-class Dylan function object with a null environment. A call-in function must appear as a C function which has no environment, so only Dylan functions which have no environment may be directly called from C.

A first-class function object in Dylan may be passed to C as a value of an opaque type even if it is actually a closure. In this case, as with all opaque types, the value must be passed back to Dylan as a parameter in a call before an operation, such as a function call, may be performed on the function.

All Dylan functions obey a consistent calling convention which checks the number and types of arguments passed for consistency with the function's contract. All first-class Dylan functions (including call-outs) will obey this contract. Call-in functions are not first-class Dylan functions, and they obey the C calling convention. Hence call-in functions are susceptible to the same level of consistency checking as any other C function, and they may rely on the programmer to ensure that they are called with appropriate arguments.

Dylan functions also support optional arguments, where the called function may determine the number of optional arguments it was actually passed. This contrasts with C's varargs mechanism, where there is no information about the number of arguments actually supplied. It is relatively simple for the FLI to map a C call-out function which accepts varargs. In this case the corresponding Dylan function object will accept optional arguments and pass them all to C. However, it's not possible to implement a varargs call-in function in Dylan without inventing a new language feature to access the arguments, because the Dylan function has no way to determine how many arguments were supplied.

Dylan functions are also capable of returning multiple values. This is a useful language feature which is often simulated in C by having functions assign to global variables (such as `errno`), or by passing the address of a variable as an argument (otherwise known as *call by reference*). For a call-in function, only the first value returned by the Dylan function may be passed back to C, because the C language can only express a return of a single value. (It is possible to define a convention to permit C code to retrieve further values, though. A C function could be specified to do this, for example). For a call-out, only one value will be returned to Dylan, unless a use of call by reference is explicitly mapped into a multiple-value return via the interface specification.

4.4 Memory Management

One important feature of Dylan is automatic memory management. Dylan provides a `make` function which allocates memory in a similar way to C's `malloc` — but Dylan has no equivalent to C's `free`. Implementations must, therefore, provide a garbage collector to recycle objects in memory which are no longer live, if memory usage exceeds available resources.

The concept of *liveness* is discussed in some detail in [G4.2.2a], and an important point to recall is that liveness is a global property of a program. Most garbage collection techniques depend upon consistent use of conventions to determine liveness — but foreign components may not obey the conventions, so global liveness information may be impossible to determine.

One way to implement garbage collection in a mixed-language environment is to use a conservative garbage collector to provide memory management services for all components, as described in section 3.4.4. A conservative collector can scan C variables and data-structures as well as Dylan ones to determine liveness. It does this by blindly scanning any area of memory which could conceivably be a *root* from which other objects might be reached, and by assuming that any value it finds during the scan which looks like a pointer to a heap object should actually be treated as sufficient proof that the heap object is live. Since such a system is able to analyse all components, it does not suffer from locality of liveness information.

Alternatively, it is possible to provide local garbage collection for Dylan components only. This can be achieved by keeping a record of all references to objects which are passed across the interface to an unco-operative foreign component. A notion of global liveness can then be determined from a combination of local information and the interface record. The interface record is, therefore, being used as surrogate liveness information for the components across the interface.

However, it is impossible to calculate liveness information accurately from the interface alone, as the interface does not provide enough information about how the foreign component uses the objects it is passed. A worst-case scenario is that an object passed across the interface will be stored in some foreign data-structure and will be kept live indefinitely — so the default action of the interface must be to record all objects passed across the interface as permanently live. This is pessimistic, and hence inefficient, in comparison with having real liveness data. Applications may exhaust their memory limits if many objects are protected from recycling in this way.

In practice, it is possible to improve the accuracy of the interface records, and so reduce the cost of garbage collection with unco-operative components. One way is to provide an implementation of `free` (or something equivalent) by which components can indicate that they have finished with an object which has been passed to them. This effectively provides graceful degradation from automatic to explicit memory management. Another technique is to remove an object from the interface record when the call which caused it to be recorded returns. This technique is only valid if the parameter passed to the called function has *dynamic extent*, (that is, if the value is not stored anywhere for a longer time than the call). It is common for parameters in C function calls to have dynamic extent, although this is normally assumed, rather than documented. (The assumption is important for C programmers too, otherwise explicit memory management would be impossible for component based programming). Ideally, the foreign language interface will allow the programmer to specify whether each parameter in a function call has dynamic extent, and will also provide a `free` mechanism for the other parameters.

Note that if a conservative collector is being used to manage Dylan components, but not the other components in the system, then it may still be advantageous to maintain the interface records. Although it is possible to imagine a conservative collector which is able to find all memory maintained by foreign allocators, and treat that memory as potential roots, such a collector is likely to be very inefficient compared with a collector which uses a smaller root set (typically, the stack, the global variables, and any statically initialized data). We assume here that interface records will always be used with Dylan components, except in the special case with a single collector providing memory management for every component.

If a local garbage collection service is being provided, then objects which are passed to foreign components must be marked as *static*, not just live. That is, they must never be relocated by the garbage collector, as the garbage collector cannot determine how to correct all references to the object from an unco-operative component. This places a restriction on the type of garbage collector which is suitable for a multi-language system. A simple copying collector, for example, *always* relocates objects after processing them [G4.2.2a]. Such a collector is, therefore, inappropriate for this type of system. However, it might still be possible to use a copying collector as part of a hybrid scheme if it is used to manage a pool of local objects only. In such a scheme, an object must be copied into a separate pool (with a different garbage collection technique) whenever it is passed across the C interface.

Conservative collectors never relocate objects which were in the root set (in case the root actually happens to be a non-pointer which is confused with a managed pointer). So for a conservative collector the static property is guaranteed for root objects, including those on the stack, and need not be explicitly marked.

The transitive closure of objects referenced by a static object do not, themselves, have to be marked as static. This is true at least if we assume that the garbage collector is only permitted to relocate Dylan objects, which are treated as opaque types in C. For a C component to access an inner reference of such an object, it is obliged to call a Dylan function, so the static property of the inner reference can be set on demand, when the Dylan function returns the reference. There is no way to enforce that a C function will not de-reference an opaque type in this way - but it is an error for a C program to do so.

4.5 Dynamic Environment

The core Dylan language provides two facilities for modifying a dynamic environment. One defines exception handlers, the other defines *cleanup* code which is guaranteed to be invoked even if there is an unusual transfer of flow of control. These facilities are described in [G4.2.2a], although the syntax has changed now that Dylan has adopted an Algol-style syntax. Under the old Lisp-style syntax, the cleanup mechanism was called `unwind-protect`.

For example, in the following code, both facilities are introduced by the `block` construct. The body of the block (which includes the calls to `start-drill-motor` and `drilling-operation`) are both executed in a dynamic environment which guarantees that the `stop-drill-motor` function will be called, even if there is a direct transfer of control to a caller of the `use-drill` function. The body is also executed in the context of a handler for `<drilling-error>` exceptions.

```
define method use-drill (drilling-operation :: <function>)
  block (return)
    start-drill-motor();
    drilling-operation();
  exception (err :: <drilling-error>)
    report-drill-failure(err);
  cleanup
    stop-drill-motor();
  end block;
end method;
```

It is only possible to raise a `<drilling-error>` exception by an appropriate call to the `signal` function in Dylan code. However, it is still possible that foreign functions may have been activated between the call to `use-drill` and the raising of the exception (for instance, if `drilling-operation` calls a C function which then does a call-in to a Dylan function which raises the exception). In this case, the intervening C function frames play no part in the handling of the exception, and are simply dropped from the stack. (This is probably the most natural semantics for raising exceptions in a mixed language environment, but other interpretations may be possible).

However, there are some more complex scenarios. Consider the case where `use-drill` is indirectly called from a C function which also calls `setjmp,` and the C function called by `drilling-operation` than calls `longjmp` to transfer control back to the caller of `use-drill`. The desired semantics are that `stop-drill-motor` cleanup function should be called — but the implementation of `longjmp` does not take any notice of any Dylan dynamic environment. There appears to be no satisfactory way of respecting dynamic environments in foreign components if C is used as the interface language. The simplest solution is for Dylan components to specify in their documentation that clients are not permitted to use `longjmp` to transfer control from a point after a call into the component to a point before. It is hoped that this restriction will not be too limiting in practice. It would be possible to provide an alternative to `setjmp` and `longjmp,` implemented in Dylan but callable from C, which would permit components to transfer control around Dylan functions.

In Dylan, exceptions can also be raised implicitly by the implementation when errors occur, such as overflow for integer addition. C code, however, is not defined to treat such conditions as errors, which effectively rules out the possibility of the Dylan exception mechanism handling this case. For the purposes of this document, we assume that each language in a heterogeneous system maintains any exception mechanism of its own independently of the other languages. As with other high-level features, it may be possible to permit the exception mechanisms to inter-operate — but only if there is co-operation across the neutral interface by all components.

5. Status of the Dylan Producer

Harlequin’s Dylan compiler has been extensively modified recently. In this section we look at some of the changes that have been made, and also at some expected changes for the future. The techniques described here were developed to implement Dylan-only applications. It is necessary to understand these techniques in isolation, before examining how they may be incorporated into a mixed-language design.

5.1 The Static Bootstrap

The initial Dylan producer made use of a dynamic bootstrapping mechanism for initializing literal data, as described in [G4.2.2b]. In essence, this meant that all literals were referenced via variables whose values were initially unset. Initialization code had to be executed the first time a Dylan program was called, to calculate values for these literals — the program then saved itself, including its initialized data, to generate a new application image. Since then, work has been progressing towards a static booting mechanism. The producer outputs the ANDF to initialize data objects statically in the same manner as a more ‘traditional’ programming language, such as C.

5.2 Use of the ANDF Extensions

Harlequin's Dylan calling convention model divides the arguments passed to functions into two different groups: *language* and *implementation* arguments. Language arguments correspond to the arguments explicitly passed to a function in the source code. Implementation arguments represent pieces of house-keeping information that are used by our implementation, and are passed "behind the scenes", without the knowledge of the Dylan programmer. Depending on the exact nature of the call, these arguments can include:

- the number of language arguments (as an integer)
- the function object being called, which contains the closure environment
- the list of next methods

Currently we have two techniques for implementing language and implementation arguments:

The first technique, which is the only one we have implemented so far, uses only caller parameters, and does not require the ANDF extensions. This has the disadvantage that it does not permit the optimization of tail-calls to jumps under all circumstances. This means that it is potentially inefficient, and is not strictly a conforming Dylan implementation.

For the second technique, which is still under development, the implementation arguments are still passed as caller parameters, while callee parameters are used for the language arguments. This permits tail call optimization in all cases, and will therefore provide a substantial improvement in terms of both execution speed and run-time stack size.

With either calling technique, our implementation provides two entry points to each function: an external entry point and an internal entry point, known as the *XEP* and *IEP* respectively. The XEP supports a general, consistent calling convention, and is used by unoptimized calls, where the compiler has no static information about the function being called. The XEP will perform argument count checking and keyword processing, before tail-calling the IEP function. The XEP always takes two implementation arguments: the function object and the number of language arguments.

The IEP accepts pre-processed language arguments — but pre-processing is only necessary when there are optional arguments, such as keywords, so often the language arguments to an IEP will be the same as they were to the XEP. All argument checking is expected to have been performed before calling the IEP, so the IEP does not require the argument count as an implementation argument. Other implementation arguments may be required, however: If a function calls next-method then the list of next methods argument will be needed. If the function is a closure then the function object argument will be required to provide the function's environment — otherwise the function object need not be passed.

Given enough static information, our compiler can optimize away calls to the XEP and call the IEP directly. This depends on the selection of the method, and the interface details of the use of implementation arguments, being statically determinable at compile-time. The programmer has some control over the amount of static information available to the compiler by using language constructs such as sealing (see [DIRM94] for more details of sealing generic functions and classes). For those cases when a function can be called directly through the IEP, a function call in Dylan will be no more expensive than a call in C. Indeed, given callee parameters for the language arguments and a reasonably optimized ANDF installer, a Dylan call could actually be more efficient than an equivalent C call because of the benefit gained from tail-call optimization.

The ANDF extensions for Dylan also include support for signalling and handling exceptions [G4.1.2]. So far, Harlequin's Dylan implementation only has support for explicit signalling of exceptions by the programmer, and we have not yet tested or used the ANDF extensions related to implicit exception handling.

5.3 Garbage Collector Implementation

Our current run-time system for the Dylan producer uses a *conservative* garbage collector, which requires minimal compiler support [G4.2.2a]. This collector manages the memory for Dylan components only (so interface records are required). Because the collector is conservative, it never relocates objects which are referenced via the roots. This removes the necessity to declare local variables as `visible` in ANDF, as would be necessary for a total collector [G4.2.2a]. We have not yet used a *total* garbage collector (which would require the compiler to identify local variables) — and we currently have no commercial requirement to do so.

Although this document shows that it is possible to use a total garbage collector and still achieve mixed language support in ANDF, we have no plans to test this. The token libraries used by the producer do not, currently, support the specification of the compiler co-operation required by a total collector.

6. Details of the Dylan Foreign Language Interface

In this section, we look at how the Dylan FLI is specified, and how it is implemented by the Dylan producer.

6.1 Interface Specification

It is intended that C interfaces will be specified in Dylan by means of an interface language called *Creole*. Creole is a language extension for Dylan proposed by Apple Computer, and it is hoped that it will evolve into a portable standard. It allows the programmer to directly import a C header file (.h) and have the compiler perform the necessary API and ABI conversions to provide the Dylan language interface. There are facilities to control what is imported from a header file, name changes of imported functions, how data type mapping is performed, etc.

We do not yet have an implementation of Creole. Testing of inter-language working with Dylan so far has been performed with low-level interface information built-in to the Dylan compiler as primitives.

The language construct that is used to import C code with Creole is the `define interface` form. The code to create an interface to the standard C input and output functions would look something like this:

```
define interface
  #include “stdio.h”;
end interface;
```

For more refined control, it is possible to selectively import only certain definitions - just some of the file handling routines for instance:

```
define interface
  #include “stdio.h”,
    import: { “fopen”, “fclose”, “fgetc”, “fputc” };
end interface;
```

Creole also allows specification of how to handle renaming on a global or individual basis:

```
define interface
  #include "stdio.h",
    prefix: "C-",
    import: { "fopen", "fclose", "fgetc",
              "fputc" => file-put-character };
end interface;
```

This makes `fopen`, `fclose` and `fgetc` available to dylan as `C-fopen`, `C-fclose` and `C-fgetc` respectively, and explicitly renames `fputc` as `file-put-character`. Name translations can also be handled by Dylan functions, so more sophisticated techniques are possible.

As well as giving control over how C and Dylan types map with respect to one another, Creole provides a convenient set of standard mappings for common types: `int` to `<integer>`, `char` to `<character>` etc. To explicitly ensure that a C `double` is mapped to a Dylan `<double-float>`, we can specify the following within a define interface form:

```
type: { "double" => <double-float> }
```

6.2 Architecture Neutrality

The Creole interface language is portable in the sense that it permits Dylan to be mapped to C, which is a portable language. However, C may be used in a non-portable manner, and it is common for C header files to include implementation-specific information along with a portable interface definition. ANDF producer output for any source code which is compiled with respect to such a header file will therefore not be architecture neutral in the ANDF sense.

Although this is not strictly a problem related to inter-language working, it is worth noting that Dylan's Creole interface could benefit from the techniques used by the C producer to minimize non-portability. The C producer permits the interface details to include architecture neutral information specified in terms of ANDF tokens, using the `#pragma token` syntax.

Since Creole parses C header files directly, it should be possible to extend Harlequin's implementation of Creole to support the same syntax as the C producer. This would significantly improve the architecture neutrality of Dylan components with respect to standard APIs. We are still designing the implementation details of Creole — but it is anticipated that the Dylan producer will at least partially support `#pragma token` within the lifetime of the GLUE project.

6.3 Calling Functions Across the FLI

Function calls in C and Dylan use different calling conventions, so the FLI must map the ABIs for each language. For each function that is called across the interface barrier, the compiler generates a stub function to map between the calling convention of the calling language and that of the called language. This function is located in the Dylan component, but is implemented as though it spans the interface.

The normal convention for passing language parameters to a Dylan function will be to use callee parameters (although our compiler does not do this, currently). The FLI must, therefore, ensure that parameters passed in a call-out and parameters received in a call-in follow the C convention and are passed as caller parameters. Internally, our compiler represents these usages of the C convention specially in its internal data structures by using specialized subclasses of the normal *lambda* and *combination* nodes which are used to describe functions and calls to functions, respectively.

Low-level language extensions have been defined to correspond to these nodes, and the Creole macros will ultimately expand into usages of these primitive extensions. The following simple example code defines a stub function suitable for a call-in:

```
define c-entry-point "simple_test_call_back" ()
  // Now we're on the Dylan side of the barrier, call the real Dylan function
  simple-test-call-back();
end c-entry-point;
```

`define c-entry-point` is syntactically similar to `define method`, except that the method name is specified as a string, to permit C's naming conventions. It does not create a first-class Dylan function, but, in this example, the interface function `simple_test_call_back` is a stub to the corresponding first-class Dylan function `simple-test-call-back`,

Similarly, the following code illustrates a simple example of a first-class Dylan function which is a stub which performs a call-out: to the C function `simple_test_call_out`, passing no arguments:

```
define method simple-test-call-out ()
  primitive-call-c-entry-point("simple_test_call_out", #f);
end method;
```

`primitive-call-c-entry-point` is the mechanism by which the call to C is made (see section 6.7.4)

Dylan functions also require implementation parameters, to represent the dynamic information necessary for a Dylan function call. For a call-out these parameters are not passed. In the `simple-test-call-out` example, above, the XEP accepts and checks the implementation parameters according to the normal Dylan conventions, but no implementation parameters need to be passed to the IEP. Normally, a Dylan call to the stub function for a call-out would be inlined anyway, with the caller making the call to the C function directly because of the compiler's standard optimizations. This means that the technique of defining the interface via stubs does not, typically, cause any speed penalty.

For a call-in function the stub is not a first-class Dylan function, hence no implementation parameters are needed when a call-in is made. When the call is made by the stub to the real Dylan function, the compiler arranges to pass appropriate parameters in the same way that it would for any other call.

6.4 Mapping Data Across the FLI

In general, all data passed across the FLI must be mapped to convert from a Dylan representation with dynamic type information to a C representation without, and vice-versa. The only exception to this is that it is possible to pass Dylan values directly to C without mapping — in this case, the value is seen as an opaque type to C. Types are mapped from Dylan to C for arguments of a call-out, and for returned values of a call-in. Similarly, types are mapped from C to Dylan for arguments of a call-in and for returned values of a call-out. Types are also mapped in this way when accessing fields of foreign structures and arrays.

Data mapping actually occurs in two stages, from *Dylan representation* to *intermediate representation* to *C representation* (or backwards). The intermediate representation is a valid Dylan representation with values which are first-class Dylan objects. It corresponds to the smallest possible amount of mapping from the corresponding C value which is still consistent with a valid representation, and is controlled by Creole directly. The intermediate representation is mapped to the Dylan representation by using generic functions which may be specialized by the programmer. This permits a higher level of control than the type mappings specifiable in Creole.

The FLI implementation defines Dylan classes which correspond to the core C types. These classes identify the intermediate representation and the corresponding mapping to the C representation. The core types are sub-divided into value types and pointer types.

The basic core pointer class is `<c-pointer>`. All intermediate representations for pointers are instances of this class, and contain the raw C pointer in a slot. A number of sub-classes of this class are provided to represent specific pointer types, and programmers are permitted to define their own sub-classes. Useful ones provided by the implementation include: `<c-function-pointer>`, `<c-char*>`, `<c-string>`.

Intermediate representations for pointer types are created by calling the low-level Dylan function `%pointer`, which takes the raw pointer and the pointer type as arguments. The notation `%`*`name`* is used consistently in our implementation to denote low-level functions which are permitted to accept or return values which are not first-class Dylan objects. These functions are never visible to the programmer. `%pointer` allocates a Dylan object of the appropriate class, and initializes it's raw data slot. For example, the following Dylan code fragment creates a wrapped Dylan object from a C string, (but does not perform data conversion to create a Dylan `<string>`):

```
// convert the raw C string to an object of type <c-string>
//
let intermediate-string = %pointer(raw-c-string, <c-string>);
...
```

In the reverse direction, the intermediate representation of pointer types is converted to the raw representation using the low-level Dylan function `%pointer-data`, which retrieves the raw pointer from the intermediate representation.

The value classes are: `<c-signed-char>`, `<c-unsigned-char>`, `<c-char>`, `<c-signed-short>`, `<c-unsigned-short>`, `<c-short>`, `<c-float>`, `<c-double>`, `<c-long-double>`, which have an obvious correspondence with C type names. These classes are pre-defined in the Creole-level interface, and other value classes may be derived from them using sub-classing in Dylan. The FLI never creates any instances of these value classes — the intermediate representation for the C value types is the Dylan class `<integer>`, apart from `<c-float>`, `<c-double>` and `<c-long-double>` which are represented as `<single-float>`, `<double-float>` and `<extended-float>` instead.

Intermediate representations of the value types are created from the raw values using the low-level Dylan mapping functions `%integer`, `%single-float`, `%double-float` and `%extended-float`. The reverse mappings are performed with `%integer-data`, `%single-float-data`, `%double-float-data` and `%extended-float-data`, respectively.

The mapping between the intermediate representation and the Dylan representation is performed by the Dylan generic functions `import-value` and `export-value` which map to and from the Dylan representation, respectively. For example, the following methods on these generic functions specify mappings from the Dylan class `<character>` to the intermediate representation `<integer>`, first forwards and then backwards:

```
define method export-value
    (c-class == <c-char>, value :: <character>)
    => (intermediate-representation :: <integer>)
  as(<integer>, value);
end method;

define method import-value
    (dylan-class == <character>, c-class == <c-char>, value :: <integer>)
    => (dylan-representation :: <character>)
  as(<character>, value);
end method;
```

6.5 Maintaining the Dynamic Environment

The dynamic environment is represented as a pointer to a chain of stack-allocated frames, where each frame is associated with a single entry in the environment [G4.2.1]. This pointer is stored in a global variable which is visible to the implementation, but not the Dylan programmer.

The neutral interface does not support accessing the dynamic environment, so C functions cannot explicitly modify it. C functions could implicitly modify it by using `longjmp` to transfer control to a point before the establishment of a dynamic construct — but this is defined to be disallowed (see section 4.5). Hence, provided that the restriction on `longjmp` is followed, it is known that the dynamic environment cannot be changed between a call-out and a call-in. Therefore the value in the dynamic environment global variable at the time of a call-in will be correct, and no code is necessary in the FLI implementation to maintain consistency.

6.6 Garbage Collection

Harlequin's Dylan implementation uses a conservative garbage collector which is used to provide a local memory management service for Dylan components only. As discussed in section 4.4, this implies that the FLI must maintain interface records for each managed object passed as a parameter in a foreign call. All managed objects passed across the interface must be recorded to indicate that they are both live and static.

The FLI is able to manipulate objects which are not allocated by Dylan. In this case, these objects are not managed by the garbage collector, although such objects may be enclosed in a wrapper giving dynamic type information for manipulation by Dylan code. In this case, the wrapper is managed by the garbage collector, while the foreign data is not. If such an object is passed back across the FLI, then it would typically be unwrapped first, so the object passed to the C function is unmanaged and not recorded in the interface record.

C objects which are passed across the FLI from C to Dylan, or vice-versa, are not managed by the garbage collector (because they are not Dylan objects), so the collector need take no steps to support them. Dylan objects which are passed to Dylan by a C function, either as a parameter of a call-in or as a result of a call-out, must have been passed to C some time before — and it is assumed that the interface record will be maintained at the time the object is first passed.

For managed objects which are passed to C with dynamic extent during a call-out, the interface record may be implemented by simply ensuring that the object is available as a local variable root for the garbage collector (for example by pushing it on the stack). Since we are using a conservative collector, this is enough to ensure that the object is treated as both live and static. This requires careful implementation in ANDF, since installers are permitted to reduce the scope of local variables as a semantics-preserving optimization, and it is possible that the scope of the local variable would end before the call-out across the FLI. This installer optimization can be disabled by defining in ANDF that the variable is `visible`. In our current implementation, this is the only time that the declaration is used.

We have not yet implemented support for passing Dylan objects to C with indefinite extent, and this is probably a rare event. The design is simple, however. If an object is recorded in this way, then it should be added to a hash-table of objects. The existence of the object in the table is enough to signify that it is live. The table must also indicate that the object is to be treated as static, which is implicit for the Boehm conservative collector, but may require special allocation of the table for a more complex conservative collector. The design of the corresponding `free` function is also simple: this should remove the object from the interface table.

If a Dylan object is passed to C as the result of a call-in, then it must be treated as an object with indefinite extent, and inserted in the interface table as for call-out parameters with indefinite extent.

If a total collector were being used instead, then it would be necessary to explicitly set the static property of objects in the interface records, in some algorithm-dependent manner. In addition, it would be necessary to preserve the chain of live variables in a global variable, as with the dynamic environment, and as described in [G4.2.2a].

6.7 An Example Interface

The following simple example shows how the C functions `fopen` and `fseek` are mapped into Dylan.

6.7.1 C Header File

Here is the relevant section from `stdio.h`:

```
typedef struct
{
  int            _cnt;     /* number of available characters in buffer */
  unsigned char* _ptr;     /* next character from/to here in buffer */
  unsigned char* _base;    /* the buffer */
  unsigned char  _flag;    /* the state of the stream */
  unsigned char  file;     /* UNIX System file descriptor */
} FILE;

FILE *fopen(char * filename, char * mode);

int fseek (FILE * stream, long offset, int ptrname);
```

6.7.2 Dylan Interface Specification

The following code must be written by the programmer:

```
// This is the actual Creole interface specification.
// It creates the interface functions which ultimately call C,
// and explicitly maps the low-level C type char* into the intermediate
// representation <c-string>.

define interface
  #include “stdio,h”,
    prefix: “c-”
    import: {“fopen”, “fseek”, “FILE *” => <c-FILE*>},
    type: {“char *” => <c-string>};
end interface;
```

6.7.3 Dylan Primitive Interface

The interface specification actually macro-expands into the following primitive Dylan code:

```
// Create the C type for FILE* pointers. This appears as an opaque type
// to Dylan, because the struct type FILE was not imported in Creole
//
define class <c-FILE*> (<c-pointer>)
end class;


// Create the stub function for calling fopen
// Note that the Creole implementation provides default mappings
// between <C-string> and <string>, including methods on import-value and
// export-value.
//
define method c-fopen
   (filename :: <string>, mode :: <string>) => (result :: <c-FILE*>)

  // Map all the arguments to the intermediate representation
  let filename_ir = export-value(<c-string>, filename);
  let mode_ir     = export-value(<c-string>, mode);

  // The real call into C
  let res_raw = primitive-call-c-entry-point
                  (“fopen”, <c-FILE*>,
                   %pointer-data(filename_ir) :: <c-string>,
                   %pointer-data(mode_ir)     :: <c-string>);

  // Return the result, after mapping through both stages
  import-value(<c-FILE*>, <c-FILE*>, %pointer(res_raw, <c-FILE*>))
end method;


// Create the stub function for calling fseek.
//
define method c-fseek
   (stream :: <c-FILE*>, offset :: <integer>, ptrname :: <integer>)
   => (result :: <integer>)

  // Map all the arguments to the intermediate representation
  let stream_ir  = export-value(<c-FILE*>, stream);
  let offset_ir  = export-value(<c-long>, offset);
  let ptrname_ir = export-value(<c-int>, ptrname);

  // The real call into C
  let res_raw = primitive-call-c-entry-point
                  (“fseek”, <integer>,
                    %pointer-data(stream_ir)    :: <c-FILE*>,
                    %integer-data(mode_ir)      :: <c-long>,
                    %integer-data(ptrname_ir)   :: <c-int>);

  // Return the result, after mapping through both stages
  import-value(<integer>, <c-int>, %integer(res_raw))
end method;
```

6.7.4 Glossary of Low-Level Functions and Primitives

%integer raw-integer => <integer> [Function]

Converts a C integer to a Dylan integer. Currently this is implemented as a tagging operation (a shift). In the future this may be a boxing operation instead.

%integer-data <integer> => raw-integer [Function]

The inverse of %integer. Converts a Dylan integer to a C integer. Currently this is implemented as a shift. In the future, this may be an indirection instead.

%pointer raw-pointer, wrapper-type => wrapper-type [Function]

Converts a C pointer to a Dylan object of the supplied wrapper type.

%pointer-data wrapper-type => raw-pointer [Function]

The inverse of %pointer. Converts a wrapped pointer into a raw pointer using an indirection.

primitive-call-c-entry-point <string>, result-type, **#rest** arg-type-pairs [Primitive]

Primitive for calling C functions. The first argument is the name of the C function to call, as a string, the second argument is the result type. The remaining arguments are the raw values to be passed to the C function, along with their associated C types. The raw result from the C function is returned. This primitive maps directly onto ANDF’s `apply-proc` construct.

7. Conclusions

The reusable component model of software development makes it possible for each component to be implemented in any language provided that the component conforms to the constraints of some chosen neutral interface language. In practice, C is often chosen as a neutral language — although higher-level interface languages exist and may be more appropriate for some programming domains.

It is possible to combine Dylan code with components written in other languages by using C as the neutral interface. Dylan may also be combined using a higher-level interface language, as ultimately the higher-level interfaces are typically defined in terms of C. When Dylan is used in a mixed-language system in this way, the other components may be used without modification, provided that they follow some simple rules of valid component design. There is therefore no cost to foreign components when they are combined with Dylan components.

If a system contains several Dylan components, as well as foreign components, then the Dylan components may be combined with each other using Dylan as the private interface language. This forms a group of Dylan components which communicates with the foreign components across the neutral interface.

The neutral interface language does not support the dynamic features of Dylan, so these features are hidden at the interface. Within a Dylan group of components, however, full use may be made of all Dylan features. The responsibility for maintaining these dynamic features around interface calls and preserving the mapping between Dylan and C belongs to the Dylan *FLI* (Foreign Language Interface). This acts as a barrier between Dylan and C code.

In order to maintain Dylan's dynamic environment for a Dylan group in a heterogeneous system, it is necessary to restrict the use of `longjmp` (or equivalent mechanisms in other languages). `Longjmp` may not be used to transfer control past any Dylan function invocations — although a higher-level Dylan mechanism may be used instead. If this restriction is followed, Dylan and foreign function invocations may be freely inter-mixed. This restriction is considered by us to be normal for good component design, anyway.

Dylan, and some other languages, have their own exception models with the ability to establish exception handlers for dynamic contexts. These models can only be unified if they are combined with an interface language which has high-level features to support dynamic exceptions. C considers only static handlers for signals, and does not support these features, so only exceptions which occur on the Dylan side of the interface can be detected and handled by Dylan code.

The Dylan interface to C may be specified using the Dylan extension language *Creole*. It may be implemented with a relatively small set of primitives which are ultimately implementable either in ANDF or in other compiler output formats. The *token interface* extensions used by the C producer (with the `#pragma token` construct) may be appropriate for inclusion into Creole, leading to architecture neutrality of compiler output.

The overhead of the Dylan code to implement the interface is usually minimal, depending on the amount of data-conversion required. Data conversion is necessary to convert between Dylan and C representations, and is dependent on details of the Dylan implementation. The cost is not related to ANDF. It is fundamental to the design of the Dylan language that data must be represented differently from C.

The need to support garbage collection in the presence of unco-operative components leads to some constraints on the design of the garbage collector, and the implementation of the FLI. Harlequin's design uses a conservative collector, which minimizes the complexity of the interface. The garbage collector restrictions are necessary for mixed-language integration in general, and are not imposed by ANDF specifically.

Dylan may also be combined with other high-level components via the same type of interface, although it may be advantageous to combine them with a higher-level interface instead. In principle, even two complex sub-systems such as garbage collectors can both be made to work without hinderance to each other providing that they both follow some simple guidelines.

All the costs and restrictions mentioned above are related to mixed-language integration of Dylan in general. ANDF imposes no barriers, and adds no direct cost, to the support of this integration. ANDF's role is purely beneficial, by the nature of its architecture neutrality.

8. References

[BW88]	Boehm, H-J., Weiser, M., “Garbage Collection in an Uncooperative Environment”, Software Practice and Experience, vol. 18, #9, September 1988
[DIRM94]	Apple Computer, *Dylan (TM) Interim Reference Manual*, Browsable on the World Wide Web through http://www.cambridge.apple.com/dylan/dylan.html, April 1994
[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
[G4.2.2a]	Tony Mann, *Initial Evaluation of TDF Support for Garbage Collection*, GLUE deliverable 4.2.2a, Harlequin Ltd., December 1993
[G4.2.2b]	Ian Piumarta, *Description and Evaluation of the Initial Dylan Producer and Runtime Support*, GLUE deliverable 4.2.2b, Harlequin Ltd., May 1994
[TDFspec]	DRA Malvern, *TDF Specification, Issue 2.1*, June 1993
[Wilson]	Paul R. Wilson, *Uniprocessor Garbage Collection Techniques*, University of Texas, 1992