CHANGE HISTORY

This is the first version.

TABLE OF CONTENTS

1. Purpose

As part of its involvement with ESPRIT project 6062 OMI/GLUE, Harlequin Ltd. has undertaken to demonstrate that TDF is powerful enough to support Lisp - a family of languages with dynamic features not found in Algol derived languages such as C. Ultimately, this work will lead to an implementation of Dylan (a Lisp dialect designed by Apple Computer Inc.) including a Dylan to TDF producer.

Harlequin has experience in implementing Common Lisp using native code compilers for a number of architectures. The initial part of Harlequin's work with OMI/GLUE has been to learn about TDF, and determine which native code implementation techniques do not map directly onto TDF. This process has been intermingled with discussions with DRA (the originators of TDF) about the most appropriate way to improve the mapping - either by making changes to the Lisp compiler technology, or by making changes to TDF itself.

This document is deliverable number 4.2.1 for the OMI/GLUE project. It contains the results of the study of those features of Lisp which are not present in static languages and might need support from TDF. The work describes how Harlequin's existing Common Lisp implementations allow both efficient compilation and an efficient implementation of the runtime system.

The study is an essential prerequisite for the ultimate goal of a Dylan to TDF producer. It is designed to be useful to both DRA and Harlequin, and will influence a future enhanced specification of TDF as well as modifications to Harlequin's implementation techniques.

This work was sponsored by the Commission of European Communities.

2. Relationship with other work

This study has provoked a number of interesting discussions with DRA about how to compile Lisp to TDF, and discussions are continuing in this area.

The work has directly contributed to the new version of the TDF specification document (which DRA expect to make available in April 1993). This specification will include some new ideas presented in a paper *Parameter Mechanisms* by Ian Currie, which addresses many of the problems described in this work. This paper is an evolving document, and Harlequin will eventually embellish it with a description of how they intend to use the new features of TDF for a Dylan producer.

3. Summary

We describe here the features of the Lisp family of languages which will not have been encountered in the design of other TDF producers. Techniques for implementing these features in Harlequin's *architecture specific* (AS) Common Lisp compilers are given. Where possible, *architecture neutral* (AN) TDF solutions are also given, with comments on their expected efficiency. Some AS solution cannot be described in TDF as it stands. We do not attempt to come to any conclusions here about what features TDF must provide - that is left to other discussions.

The dynamic features of Lisp can be divided into a number of categories. These include dynamic type checking, garbage collection, dynamically sized parameters during function calls, closures of functions,

language interactivity, and dynamic loading of code and data. None of these will have been encountered by implementors of C to TDF producers, and all present some potential difficulty for mapping onto TDF.

Harlequin's Lisp implementations rely on carefully chosen function calling conventions to implement most of these features efficiently. It is expected that a suitable level of support for tailoring calling conventions within TDF will enable most of the dynamic capabilities. In addition there will be a requirement for dynamically loading and relocating code and data.

4. Type system

4.1 Tagging

Lisp is dynamically typed, which means that type information must be carried with each object. Conventionally in Lisp systems, this has been done by *tagging* pointers to objects. This involves using some of the bits of the pointer to contain information about the type of object. Further type information may be stored inside the object itself. Since tagging is so widespread in Lisp implementations, and TDF now offers support, I will not attempt to describe any other possible dynamic typing mechanism.

From an efficiency point of view, the type of tagging scheme used can be extremely important. A good illustration of this is the implementation of small integers in one of Harlequin's tagging schemes: here, a small integer is not represented as a pointer to an object; instead it is represented as an immediate value in twos-complement form, with the bottom two bits (the tag bits) always zero. Hence the Lisp integer *i* is represented in the same way as the C integer *4xi*.

Using hardware add or subtract instructions on 2 tagged integers yields a tagged integer. Hence addition and subtraction of small Lisp integers may be performed by using the appropriate hardware instructions directly, with no speed penalty compared with a C implementation. This is true for all the standard bit operations too (or, and, xor etc.)

Multiplication and division are not quite so simple. Simple hardware multiplication of 2 tagged integers yields a result with the bottom four bits all zeros. It is easy to see that the correct results can be obtained by untagging either one of the integers for multiplication, or untagging the divisor for division. Hence the penalty for tagged integer multiplication or division is one untagging operation (shift).

If the only operations allowed on architecture neutral tagged operations are tag and untag, then the simple arithmetic operations require both operands to be untagged before the operation, and the result to be tagged after the operation. On a RISC machine an addition would now take 4 instructions compared to 1 instruction for an AS implementation.

Since arithmetic operations on integers are among the most common of operations in any computation, a factor of 4 efficiency loss for both code size and speed would have a major impact on the overall system efficiency.

4.2 Bignums

Many Lisp dialects (including Common Lisp) support arbitrary precision integers in a seamless manner. A common implementation technique is to represent small integers (`fixnum`s) as immediate values using a tagging scheme similar to the one described above. Integers which are too large to be represented as imme-

diate values (*bignum*s) are instead represented as a pointer to a block of memory containing as many words as are necessary to describe the integer. This hybrid approach has the advantage that small integers (which are by far the most common) require no heap storage and can be manipulated efficiently. Of course, manipulation of bignums can be much slower than corresponding operations on fixnums, but this seems unavoidable.

The problems of small integer arithmetic are made even worse when we start considering overflow. Consider, for instance, the addition of two fixnums. It is normal for a Lisp compiler to generate code for an overflow test, because overflow is not an error in Lisp - instead it indicates that a bignum must be created. In the AS scheme shown above, the normal processor flags may be tested to determine if overflow has occurred (with an overhead of one conditional branch in the common no-overflow case). This may not be true for the AN solution, where the arithmetic operation is applied to untagged numbers. In the AN case, an overflow test would presumably be implemented as a range test (with an overhead of 2 comparisons and 2 conditional branches for the common case).

4.3 Float types

Some Lisp dialects (notably Dylan[1]) require that the implementation of floating-point numbers is IEEE compliant. Dylan actually defines 3 classes of floating-point numbers: `<single-float>` `<double-float>` and `<extended-float>` which are defined in *[Dylan]* to implement the IEEE standard floating-point formats. Inherently, this is an architecture specific definition, and will require specific TDF support.

4.4 Dynamic creation of types

Common Lisp contains a powerful type definition language, and includes features for creating new types dynamically. It should be possible to implement this mechanism in terms of simpler Lisp primitives, and so there is no requirement for any extra TDF features to support this.

5. Garbage Collection

There are many ways to implement garbage collection (*GC*), and the specification of TDF has left GC to the language implementor. The exact design of GC may have an influence on the extra TDF tokens required to implement the language.[2] Some ideas are central to all GC strategies, and hence there are certain primitive ideas which it must be possible to express in TDF.

The purpose of GC is to reclaim the space used by objects which a program no longer requires, so that the space can be reused. Normally, GC is invoked automatically when an attempt has been made to allocate an object and there is insufficient space in the allocation area. The assumption made by most GC algorithms is that if a program has no means of accessing an object without breaking the language model, then the object can be treated as garbage, and its space reclaimed. Hence a central part of any garbage collector is the ability to find all *live* objects so that everything else can be reclaimed.

1. Dylan is still not a fully defined language, and there is still ongoing discussion amongst interested parties about whether floating point numbers should be made optional in the language.

2. E.g. a generational garbage collector may require a special token to be used at each store into memory. This would flag the new pointer location as an extra GC root, so that the possibility of having created a pointer from an old to a new generation does not break the generational model.

5.1 Finding live objects

Most recognized GC techniques find all live objects by looking first at a carefully selected set of *root*s. This set is chosen by the language implementor so that it references the same set of objects as the program. In the case of Common Lisp, the roots would normally include variables on the Lisp stack[1] and the list of symbol packages.

Once an object has been encountered during GC, the processing which must be performed is algorithm dependant. A common requirement, though, is to identify all the references from the object to other objects, so that the other objects can also be processed. To do this, the garbage collector must recognize which fields inside an object refer to other objects. For example, if an array is encountered, the garbage collector must determine both the size and the type of the array before identifying what it refers to; if the array is an array of bits, the binary numbers in the array will not be valid pointers, but if it is a general purpose array then each array cell may reference another object.

With the exception of the stack, it seems that this should not pose any particular problems for TDF; the requirement for dynamic typing means that we must already be able to identify the type of any Lisp object. Finding all other references from a stack, however, is a much harder problem.

The stack is a record of the current state of the program, and includes information such as function return addresses, arguments to functions, local variables (some of which should be processed during GC, and some of which should not), and dynamic information associated with the Lisp language features described below (e.g. `unwind-protect`, `catch`, and `special` binding). Many Lisp implementations will allocate objects on the stack, too; these objects require special processing during GC.

5.1.1 Harlequin's stack frame implementation

Harlequin's Lisp implementations rely on scanning the stack from top to bottom during GC and identifying what type of data is found at each location. The stack space associated with each function invocation is distinguished by means of a saved frame pointer, linking each function frame together. For some languages, building a stack frame in this way may be necessary for particular types of function only. In Harlequin Lisp, it is obligatory for every function unless the compiler can be certain that the function can never allocate memory, even in an error situation[2]. This normally means that the stack frame can only be avoided if the function is a *leaf case* (i.e. it calls no other function).

While processing each frame, Harlequin's garbage collector first looks at the data at known locations within the frame to get information about the called functions. The intention is to find all the active Lisp function objects, since these are clearly live data, and should not be treated as garbage. In our implementation[3], part of the convention of building a stack frame is to push the calling function's *constants vector* onto the stack; this contains all the function's static references to other Lisp objects, including a pointer back to the function object itself. The return address is at another fixed offset from the frame pointer, this is ignored during GC (except in the case where the function is moved during GC, in which case the return address must be fixed up).

1. Or stacks, since many Lisp implementations offer a multiple thread implementation.

2. Note that calling `error` is very likely to allocate, since this will invoke the debugger unless the error is handled. Once inside the debugger, any Lisp function may be invoked,

3. I am actually describing a hybrid of our CISC and RISC implementations, choosing the most AN features of both. This mixture might be confusing to those familiar with Harlequin's implementations, but is hopefully clearer otherwise.

The constants vector contains more than just the constants referenced by the function - it also gives information to the garbage collector about how the function uses the stack. This data includes how many registers have been saved in the stack (since these must be processed during GC), how many variables there are which need processing, and how many variables must not be processed[1]. Any data on the stack which is not in a location described by the constants vector must be scanned. Any data which requires special processing (e.g. catch or binding frames) is marked by a stack value which is tagged to represent the type of processing required; the tag is chosen so that it cannot be confused with a normal Lisp object. If the garbage collector encounters such a tagged value while scanning the stack, then the data is processed in a suitable manner before resuming the scan. For example, a binding frame comprises a pointer to the next binding frame (ignored by GC), a Lisp symbol and its previous value (both processed by GC). If the garbage collector encounters a pointer object which is not tagged for special processing, the object is processed normally like general Lisp pointers.

5.2 Reclaiming dead space

The mechanism for reclaiming the space of objects which are no longer live is very dependant on the GC algorithm. *[B92]* describes some of these techniques, and the problems of implementing them in an AN manner.

5.3 Compaction

One important requirement of GC is to compact data. Any algorithm which does not do this is doomed to failure from fragmentation problems, unless objects are of fixed sizes (not true in Lisp!). A necessary consequence of compaction is that objects move in memory, and references between them must be fixed up.

The mechanics of locating the addresses to be fixed up because an object has moved are a problem for the garbage collector, and I won't discuss them here. But the code produced to implement the language itself must make it possible for GC to fix up references after compaction, and TDF must provide any necessary primitives.

One such primitive concerns position independence of code. Since code may be dynamically created in Lisp, it must be garbage collectable (and hence compactible) too. Hence TDF must be able to generate code which is either relocatable or contains enough information that the garbage collector can locate absolute pointers within the code or relative pointers to outside the code. Note that there are also pointers into code from the stack, which contains return addresses. We have already seen how the garbage collector must be able to locate return addresses and function objects, so this should not cause any particular new problems.

Another problem associated with the relationship between GC and code is that the constants might move. As has already been seen, Harlequin's Lisp implementation considers functions to be a tuple of the code and the constants of the function, effectively making a static closure. This technique makes relocation of the objects very easy, as the code references the constants indirectly through a register which points to the constants vector. I see no particular reason why this cannot be true for a TDF implementation too, although a pre-requisite is that there must be some control over the calling sequence so that the code can always find its constants.

1. Sometimes a function can be compiled more efficiently if it uses temporary variables which are not valid tagged pointers. These variables would not normally be live over any part of the function which might provoke a garbage collection. There may still be space reserved on the stack for the variable even when the variable is not live, and this data must not be processed by GC.

6. Functions

Integer arithmetic and GC have already been mentioned as important factors in determining the efficiency of a Lisp implementation. Of these, GC is most likely to affect the run-time performance. However, if there is one single feature that is likely to have the most influence on the overall efficiency of any language (both speed and code size), then it must be the overhead of a function call.

As we will see below, function calling and returning in Lisp have properties not found in C. The quality of implementation of these properties will have a major impact on the quality of the implementation as a whole.

6.1 Parameter passing

As a dynamic language, the number of arguments passed to a Lisp function may be decided dynamically. Language features (such as `funcall`, `apply` and the ability to dynamically redefine functions) make it impossible for a compiler to check that a function is called with an appropriate number of arguments[1]. It is the responsibility of the called function to check the validity of the arguments passed, and to flag any errors by using the standard mechanisms (e.g. by signalling a condition with the function `error`, in the case of Common Lisp).

A lisp function describes its argument interface by means of a *lambda list*. Special syntax for the lambda list allows the function to handle arguments in a variety of ways, both positional and by keyword.

Common Lisp provides the following as lambda list qualifiers: `&optional`, `&rest` and `&key`[2]. Formal parameters which are not qualified are mapped to arguments in a positional manner, like in C. These are called *required* parameters, and it is an error if an argument is not supplied for them. `&optional` parameters are also mapped positionally, but as the name implies, it is not obligatory to supply the argument. Only one `&rest` parameter is allowed; if it is given, then any arguments passed from that position onwards are collected together to form a list[3]. The `&rest` parameter is initialized to this list. Like `&rest`, the `&key` qualifier also allows an arbitrary number of arguments to follow[4]. Any arguments passed from the position of the `&key` onwards are processed as keyword-value pairs. If, during processing, a keyword is found which matches any of the parameters qualified with `&key` then the corresponding value is used to initialize the parameter. If arguments are not supplied for `&optional` and `&key` qualified parameters then a default initialization will be performed (usually this means setting the value to `nil`, but the programmer may specify any initialization).

So some functions expect a fixed number of arguments, while other functions can do complex manipulations with an arbitrary number of arguments. How can this be implemented in an efficient way? We actually have two schemes in Harlequin, but they share many features.

1. However, a compiler may assume that certain low-level functions specific to the implementation are only ever called in a controlled manner. In this case it may be acceptable to optimise away the error checking inside the function. It may also be possible to optimise the call.

2. Common Lisp also accepts the qualifiers `&aux` and `&allow-other-keys`. Since these do not affect the mapping of the arguments passed to the function, they are not described here.

3. Dylan provides a similar facility with the qualifier `#rest`, but defines that the arguments are collected to form a sequence. This gives implementors more freedom than with Common Lisp, as it allows the collection to be either a list or a vector.

4. `&rest` and `&key` may both be supplied. In this case, the collection to form a list and the keyword processing are both performed on the same set of supplied arguments.

6.1.1 RISC calling mechanism

Harlequin's Common Lisp for RISC processors adopts a calling sequence which is optimised towards keeping information in registers.

The first few arguments are moved into registers[1]. If any more arguments are required they are pushed onto the stack in evaluation order (i.e. left to right). Note that this ordering reduces the number of temporary variables which are necessary to construct the call, since each argument can be pushed onto the stack as soon as it is evaluated[2]. When all the arguments have been initialized, the caller puts the total number of supplied arguments into the *arg-count* register, as an untagged integer. Finally, before the call, the Lisp object representing the destination function is moved into the *function register*. This function object is composed of two parts: *code* and *constants*. The code is a tagged Lisp object containing the binary code to implement the function. The constants are a Lisp vector of the non-parameterised data manipulated by the code.[3] The call itself then involves loading the code from the function object, and jumping to the first instruction. On return from the call, the caller pops any arguments it may have pushed onto the stack.

The called function processes arguments depending on its lambda list. If the function has required parameters, then a check is made to ensure that enough arguments were supplied. Unless there are `&rest` or `&key` parameters, a check is also made that there were not too many arguments passed. If either of these tests fails, then a tail-call is made to a function which will signal the error. This tail-call uses a special convention in order to reduce space (since almost all functions will have to contain code for this[4]). The convention is simply the bare minimum of a jump to the code of the handler[5] since the rest of the information (the number of arguments, the arguments themselves and the destination function) may all be found in registers.

The processing of optional parameters requires testing the arg-count to see if an argument was supplied. The parameter is initialized to the argument if applicable, otherwise it is initialized by code in the function. Where there is more than one optional argument, it is not necessary to test the arg-count for each parameter once the limit of supplied arguments has been reached.

Processing of any `&rest` parameter potentially requires a large amount of computation to create the list. This is therefore performed out of line by calling a dedicated function. This dedicated function is then required to map over the arguments of its caller to create a list of the appropriate values.

Keyword processing is handled by always creating the collection of extra arguments (even if there was no `&rest` parameter), and searching for each key parameter in the collection list.

Certain low-level functions are compiled at a low safety level, and do not ever check the number of passed arguments. If the compiler knows that a called function is unsafe in this way, then it may compile a call to the function without setting the arg-count register. Note that this optimisation is a simple subset of the standard calling convention. An unsafe function called in the normal way would still interpret its arguments correctly.

---

1. Typically 4 registers are used for argument passing, depending on the processor.

2. A reverse ordering might possibly have some advantages for the callee, but would be a disadvantage for the caller.

3. This includes Lisp objects which appear as constants in the function. It also includes information such as the name and lambda list of the function for debugging purposes.

4. Many functions also check for stack overflow. To reduce code size, the same error handler is called in this case. The handler will check again for stack overflow to determine why it was invoked.

5. This jump must be before the stack frame of the function has been built for it to be most efficient. Hence the argument count checking must also be before building a stack frame.

At any point in the compiled code of a function, the compiler may choose to build a stack frame. The compiler will normally optimize this so that a conditional branch of the program which can operate without a stack frame is able to do so without the expense of creating one. The stack frame will never be built before the argument count check and stack overflow check. Building a stack frame involves: pushing the return address onto the stack; pushing the caller's frame pointer register and setting the new frame pointer to the stack pointer; pushing the caller's constants vector (in the constants register) and loading the new constants vector into the constants register with an indirection from the contents of the function register; pushing any callee-save registers which are used in the function; pushing and initializing (with a safe value for GC) a spill area for those temporary variables which cannot be located in machine registers.

6.1.2 CISC calling mechanism

The CISC calling mechanism is currently only used for PCs with Intel 386 and 486 processors. It is worth examining this mechanism both for its similarities and its differences. The differences are partly historical, partly because the 386 has few registers, and partly because some good ideas have not yet filtered from one implementation to the other.

The mechanism of the call itself is very similar to the RISC implementation. First, all the arguments are set up: in this case all the arguments but the last are pushed on the stack in evaluation order; the last argument is always passed in the *argument register*. Next, the *arg-count* register is initialized as for RISC machines, except that here the arg-count register is only a byte[1], which is efficient to set on a 386. Finally, the details of the call itself depend on the type of function being called: If the function is a symbol (i.e. a normal call of a named function), then the symbol appears as an in-line constant in the code and the call instruction jumps to an address stored at a fixed offset from the symbol pointer. If the function is known to be a compiled function object (*CFO*), then the tagged pointer to the CFO actually gives the address of the first instruction which can be called directly. If the function type is not known at compile time, then the function object is moved into the *dispatch* register, and a dedicated dispatcher function is called, which decides how to call the intended function from its type and jumps to the appropriate address. After the call, the stack arguments do not have to be popped from the stack - this is done by the called function.[2]

If there are only required parameters, the callee checks the arg-count as for the RISC implementation. The important difference is in how the specialized parameters are processed. Since the callee has the responsibility for removing its arguments from the stack there is no restriction on the callee to preserve the number of arguments on the stack during the call.

Consider a lambda list which ends with a single optional parameter. If the caller does not supply an argument for this parameter, then the callee can insert the value of the argument register onto the stack after the other stack arguments, and move `nil` into the argument register. Apart from the arg-count, the arguments now are arranged as if `nil` had been supplied, and all arguments can now be treated in the same way as required arguments, already initialized, and in a known position in the function frame. The technique of adding `nil`s scales up to any number of optional arguments.

Similarly, any rest arguments can be collected up and the original supplied arguments removed from the stack, leaving the collection list in the argument register. Again, apart from the arg-count, the function now appears to have been called as if the rest parameter were directly passed a list, and all parameters including the rest parameter can be treated as required parameters.

1. This imposes a restriction of 255 on the total number of arguments which may be passed to a function. The definition of Common Lisp allows for such a restriction provided that the limit is not less than 50.

2. Rationale: there are more calls to functions than there are functions. Since, in general, a function will only have one return sequence, less code will be required to pop stack arguments if this is done in the return sequence, than if done after the call. There are other advantages too, from the callee's viewpoint; these are described below.

Even keywords can be treated in this way, after some rearrangement on the stack. In this case, the keyword value pairs must be shuffled down the stack, to make room for values for the each keyword parameter, as if it were a required parameter. The space so created is filled with `nil`s since this is the default value for each keyword parameter. Next, a search is made through the keyword value pairs for each keyword parameter; if it is found, the value is put into the appropriate place for the parameter in the new area. Finally, all of the shuffled arguments are removed from the stack, leaving the required set of pre-initialized arguments[1].

The advantage of treating arguments in this way is that all the manipulations described can actually be performed in a handful of dedicated functions, which can then be called as required in the prologue of functions which have complex lambda lists to process. For functions with keyword parameters, the dedicated function must be passed a vector of the keyword parameters so that the relevant searches can be made. For all parameter processing, the dedicated function must be passed the number of required parameters, and the number of optional parameters. Note that the dedicated functions can even call the error function when the number of supplied arguments is inappropriate. This reduces the code size of functions with complex lambda lists so dramatically that I expect that we will eventually adopt this technique for RISC machines too (along with the necessary change that the callee removes the stack arguments).

For this technique to work, a special calling convention for the dedicated functions is required, so that they preserve the argument and arg-count registers. The two integer arguments to the dedicated functions are carefully encoded into a single register; the keyword vector is passed in another register which is not used in the normal convention. Note that the dedicated functions must actually manipulate the stack frame of their caller, which makes them very unusual functions indeed[2].

6.1.3 Architecture neutral calling mechanism

It may be possible to call standard Lisp functions with TDF constructs using the standard mechanism `apply_proc`. One possibility would be to pass the count of the number of arguments as the first argument. This gives the functionality required for a standard call, although there are some minor efficiency concerns about the order of pushing arguments onto the stack (this is likely to be the reverse of the Lisp evaluation order).

Unfortunately, it is difficult to manipulate stack arguments in an AN fashion, so the techniques described above which use special calling conventions to call dedicated functions may not be possible. The code increase because of this could be substantial.

6.2 Apply

In section 6.1 we saw that for called functions argument handling is more powerful than in C. We also saw that all functions must be able to be called with any number of arguments. The function calls themselves, however, all used a statically determined number of arguments in a similar way to a C function call. In fact Lisp does allow a dynamic call too; in Common Lisp[3] this is called *`apply`*.

`Apply` is Common Lisp function which takes as arguments a function to be called, and the parameters to be passed to the function (an `&rest` parameter is used for this). The last argument to `apply` is a treated as a list[4] of further arguments for the function, so that the arguments for the function consist of the last

1. For simplicity, I have omitted some details here, like how keyword and optional parameters are initialized to non-nil values. This is actually straight-forward, as is the problem of handling any combination of optional, keyword and rest parameters.
2. It is not possible to code these functions in Common Lisp.
3. Also for many other Lisps.
4. In Dylan the last argument may be any sequence, e.g. a list, a vector, a string, a double ended queue etc.

argument to `apply` appended to all the other arguments to `apply` apart from the function itself. For example:

```
(apply 'some-function 1 2 '(3 4 5))
```

is equivalent to the static function call

```
(some-function 1 2 3 4 5)
```

In this example, the final list is of static size, so that the compiler could transform the dynamic call into the static call. In the general case, though, the final list will not be known at compile time.

It appears that this causes problems for TDF, since there is currently no primitive in TDF for a call with a dynamically sized number of arguments.

6.3 Returning values

Common Lisp (and Dylan) allow functions to return *multiple values*. This mechanism has been designed to be both powerful for the programmer and inexpensive for the implementor.

The only way of creating multiple values is to call the function `values`[1]. However, many constructs in the language return all the values returned by a nested construct. This includes the `progn` construct (roughly equivalent to { ... } in C), and also the return from a function. There is no reason why tail-calls should not be optimized because of this feature, since functions return all the values of a tail called function.

Very often, all but the first of the multiple values returned by a called function are ignored. In this case, if zero values were actually returned, the single required value would default to `nil`. When it is required to use all the multiple values returned by an expression, features such as `multiple-value-bind`[2] are provided in the language.

The majority of Common Lisp functions return a single value, and the majority of calls expect only a single value to be returned. From an efficiency viewpoint, these cases must be slowed down as little as possible to support those few cases where multiple values are useful.

In Harlequin's implementation, if a function returns a single value, this value will be put into the *result* register before the return. For those cases where multiple values are generated, the values will be put into a static multiple values vector; in addition, the frequently expected single value will also be moved into the result register (`nil` will be explicitly used for the zero values case).

The common case of ignoring the multiple values after a call is handled efficiently using this technique, since the value is always in the result register. When the multiple values are required, they must be retrieved form the static vector.

There is an overhead at function return time, since a function must indicate the number of values it is returning, even if this value is 1. Harlequin's RISC implementation does this by setting the *mv-count* register to the number of values. Note that this does not need to be done when a function returns the result of another function call, since it will have been done in the innermost call.

1. Compilers would normally in-line a call to `values`.
2. In Dylan, the normal binding construct `bind` provides the syntax for a multiple value bind too.

Harlequin's CISC implementation stores the number of values in a static location (which is slower to initialize as it is in heap memory). In addition, the presence of multiple values is indicated by setting a processor flag: when set, a single value has been returned, when unset there are multiple values. This is actually more efficient on the Intel 386, since (un)setting the flag is a single byte instruction, and it is unnecessary to set the static value count for the common single value case.

I believe it would be possible to implement multiple values in TDF in a similar way to our RISC implementation. Functions can return a pair of values: the single value, and the count of the number of values. Multiple values can be stored in a static vector for later retrieval when required.

TDF allows functions to return values of arbitrary SHAPE, so returning the pair is possible within the current definition. I am not clear of the mechanism by which the pair of values is returned. This will be the key to whether TDF can be efficient here. If the 2 values can be returned in registers (assuming an optimised installer) then there is no reason why TDF should not be able to implement multiple values as efficiently as Harlequin's own implementations. If not, the overhead of every function call will increase, with a corresponding impact on overall efficiency.

6.4 Closures

Many languages (including Pascal for instance) allow lexically nested function definitions, and allow the innermost function to access the variables of the parent. In Lisp (and other languages where functions are first class objects), it is possible to refer to the innermost function even after the calling frame of the outer function has been unwound[1] (i.e. the inner function has indefinite extent). Since it is possible for the inner function to refer to the variables of the parent, these variables must also have indefinite extent. Functions which refer to the variables of their lexical parents in this way are called *closures*. The variables they refer to with indefinite extent are called *closed over variables*.

Normally function variables are either stored on the stack or in registers. Closed over variables, however, must be stored on the heap. In Harlequin's implementations of lisp, these variables are stored in a Lisp vector which is created by the parent function.

In the RISC Lisp implementation, the closure is associated with the vector by means of the function's *constants*, which in this special case are actually implemented as a cons cell containing both the normal constants vector and the closed over variables vector. Making a new closure requires making a new closure object, setting the code field to the code of the inner function, and setting the constants field to a newly created cons of the constants vector and the closed over variables.

In the CISC implementation, the closure is associated with the vector by representing the closure object as a special calling function stub. This stub is a function which when called puts the closed over variables vector into the *closure register* and then jumps to the inner function. The inner function is compiled to expect to find the vector in this register. Making a new closure requires making this new function stub, and initializing the real function and the closed over variable vector as the two constants of the stub.

The AN implementation of this may turn out to be straightforward in TDF, once the problems of associating constants with first class functions have been sorted out. However I am concerned that it may prove difficult to describe how to build a closure object in an AN manner.

1. Pascal does allow indefinite scope references to procedures through procedure variables, but disallows assignment of inner procedures to variables. This restriction stops inner procedures from accessing parameters which are not in the dynamic scope of the parent.

6.5 Tail recursion

We have seen above that language features such as multiple values do not prevent optimisation of tail calls as jumps. Indeed Harlequin's Lisp implementations will optimise tail calls in this way whenever possible (including tail calls to self).

In Common Lisp, this is regarded as an optimisation (albeit a very important one). However there are Lisp dialects where the language definition indicates that tail calls must be optimised as jumps rather than calls. Scheme is defined in this way, and Apple have recently indicated that they will be changing the definition of Dylan to include the Scheme definition of proper tail recursion.

As far as I can see, it is not possible to guarantee optimisation of tail calls in TDF. Of course, one would hope that a suitably optimised installer would use a jump wherever possible, but this must be *guaranteed* for a Scheme or Dylan implementation.

A further problem is that TDF installers are currently constrained to use the C calling convention. For most C implementations, the calling convention leaves it to the caller to pop any arguments pushed onto the stack, since this allows a variable number of arguments to be passed to a function without the callee having to know how many arguments it has received[1]. Unfortunately, this probably makes it impossible to implement a tail call as a jump. Consider the case where the tail call requires adding more stack arguments; the original caller will not be expecting to remove these extra arguments, so stack integrity is not preserved. Hence Scheme and Dylan **cannot** be implemented with function calls using only the C calling convention.

6.6 Conclusion

We have seen that Lisp function calling requires a different function calling mechanism from C in order to implement tail calls properly. We have also seen that AS implementations of Lisp make use of a number of special calling mechanisms to implement parameter mapping, closure calling, multiple values etc.

It would appear that an efficient TDF Lisp implementation must have access to the function calling mechanism at a much finer grain than is currently possible.

7. Interactivity

7.1 Compilation

Common Lisp defines two entry points into the compiler: `compile` and `compile-file` (in Dylan, it has not yet been defined what standard entry points there will be - if any).

7.1.1 Compile-file

As the name suggests, `compile-file` is the file compiler: it reads in ASCII data from a file stream, and generates a binary representation in a file which may be loaded at any time. Lisp users often refer to the binary file as a *FASL* file; this is short for FASt Load, so called because loading it should have the same effect as loading the original, uncompiled, ASCII file but the results should be faster to load.

1. The C calling convention seems to be designed to support `printf`, which is one of the few functions in the language to take a variable number of arguments. Lisp has its own mechanism for variable arguments, and does not need to be tied down with `printf`.

The semantics of reading the source file and converting it to compiled code are carefully defined in Common Lisp. The source code is read with the normal Lisp reader function `read`, which converts the text into a Lisp object representation. This Lisp representation is called a *form*. As the compiler processes the form, it must expand any *macros* in the form. Macros are syntactically similar to functions in Lisp, but uses of macros are expanded at compile time, whereas functions are called at run time. If a form is a use of a macro (i.e. the first object in the form names a macro), then the form is passed to the macro expander, which converts it to a new form which is then used instead. This new form may then require further macro expansion. This macro facility is very powerful, but it obscures the debugger mapping between program statements and source code line number, since the compiler does not actually process the same source code as written in the file.

FASL files have some subtle differences from the object files produced by C compilers. A C object file contains compiled code for each named function in the file; this information is later resolved by the linker to make a static program. A Lisp FASL file typically contains many *one shot functions*, which are anonymous functions requiring no arguments, and are executed only once at the time the file is loaded. One shot functions have the side effects corresponding to the *top level forms* of the Lisp source file (i.e. the Lisp statements at the outermost textual level). These top level forms can contain arbitrary Lisp code. A common constituent of a top level form is `defun` which has the side effect of associating a function object with a symbol[1].

Linking of these FASL files is implicit and automatic, since FASL files must preserve semantics about the similarity of multiple references to certain types of object. One fundamental rule is that all references to a symbol with a particular *package* (i.e. module) and *name* are guaranteed to correspond to the identical symbol object even if the references are from different files. Since most functions and variables are named by symbols, this ensures that two portions of a program which have been separately compiled can communicate with each other.

FASL files are not only used to load compiled code into a Lisp image. Since it is possible to define a Lisp function to return any Lisp value, FASL files must be able to record arbitrary Lisp data structures too. This makes them suitable for saving data in a form that can be loaded into another image. This is often useful for saving complex structures such as hash-tables, or saving data structures containing objects of a class defined using the Common Lisp Object System (*CLOS*). The function `make-load-form` allows user extensibility to the semantics of constant similarity; this means that FASL files must be manipulated by a Lisp program, and not by TDF directly.

I expect that the choice of FASL format will be irrelevant to TDF. Our Lisp compiler will probably generate TDF function and constant descriptions in place of native descriptions, but otherwise use our existing format. The interaction with TDF must occur at load time, which we have seen must be performed in Lisp,[2] so the TDF interface need not be concerned with how TDF descriptions are extracted from the FASL file.

At present TDF provides no facilities for dynamic translation of TDF to native code. The remaining requirement appears to be a function which can be called from Lisp which takes a pointer to a TDF encoding and returns a pointer to the corresponding compiled code, or constant, suitably allocated for GC. The efficiency of this function is not of paramount importance. It might be acceptable for the function to dump

1. This one is such a common case that Harlequin's FASL format has an optimisation for it. An anonymous function is not present in the FASL file for top level forms such as `defun` or `defmacro`. Other optimisations are also performed, such as collecting together top level forms to create fewer one shot functions.

2. This leads to some interesting bootstrap problems, which are discussed in Section 10 below.

the TDF encoding to a temporary file and then fork the TDF installer to generate an object file. A means must then be found for loading the contents of this file into freshly allocated memory.

7.1.2 Compile

The other compiler entry point, `compile`, manipulates Lisp forms rather than source code. This object compiler is often used interactively to compile interpreted function definitions (e.g. after first testing them interactively).

`Compile` may also be called dynamically (like any other function). As an example, when a generic function is called for the first time with arguments of particular classes, the mechanism which implements the behaviour of the generic function may choose to create and compile a new function to call the relevant methods in the appropriate order.

It is conceivable that `compile` might be used to implement the interpreter itself. It is permitted for a Lisp implementation to compile Lisp forms before evaluating them. While the overhead of the compilation may slow down the evaluation, there may be a useful advantage in not needing a separate interpreter.

It should be possible to implement `compile` using the same TDF interface function described above for loading FASL files.

7.2 Debugging

The definition of Common Lisp includes the notion of a debugger. The meaning of the term “debugger” is actually quite loosely specified, and is simply described as an “interactive condition handler” *[S90]*. In practice, all Common Lisp debuggers offer a minimum functionality that I would like to treat as a basic specification for a “reasonable” Lisp implementation. This functionality includes full access to the Lisp interpreter, a way of displaying and invoking any *restarts*[1] which are currently active and a means of displaying a function call backtrace. In addition, the following features are desirable: a means of displaying the arguments of active functions, a means of displaying the local variables of active functions, a means of displaying special variable bindings and catch frames which are active, and a means of returning a user supplied value from any active function.

Harlequin’s debugger implementation relies on the debugger being able to scan the stack for function frames and other language constructs. This is also a requirement for GC, and so does not further complicate the use of the stack. Symbolic information about function names and variable names is stored in a fixed place on the constants vector of every function. Again, it is a requirement that GC can find the constants vector for each active function’s stack frame.

At first sight, it would appear that no extra TDF primitives are required to implement a debugger, since it is already a requirement that all the information on the stack is available to GC too. I am still unsure whether this stack information can be represented in an AN fashion. If not, then presumably both the garbage collector and the debugger will have to be written in AS code, which would reduce the final portability of the TDF solution.

7.3 Tracing and Stepping

The macros `trace` and `step` are included in the definition of Common Lisp. `Trace` is loosely specified to print the function name and arguments passed when a traced function is called. `Step` is similarly loosely specified to allow an interactive “single step” facility as a form is evaluated.

1. Restarts are described later in section 9.1, on Conditions.

It is possible to implement both `trace` and `step` in portable Common Lisp, so there should be no need for any TDF extensions to support these features.

8. Miscellaneous Language Features

8.1 Catch and throw

Common Lisp provides an associated pair of language constructs for transfer of control called `catch` and `throw`[1]. `Catch` is a special form which uses the result of evaluating its first argument as an object to name a *catcher*. The remaining forms inside the `catch` form are evaluated in order in an environment in which the catcher is available as a target for a control transfer using the `throw` special form.

`Throw` evaluates its first argument which is used to name the catcher for the transfer of control. If multiple catchers exist with the same name, then the most recent will be used. The second argument is then evaluated, and the resulting value(s) are returned from the named `catch` form. It is an error to use `throw` when there is no suitable catcher, and any implementation is required to detect this, and signal an error in the context of the thrower - i.e. the target must be found before and not during the throw. If `throw` is not used to transfer control during the execution of `catch`, then the result of the catch is the result of evaluating the last form inside the `catch`. The catcher is disabled when the `catch` form returns its values.

Harlequin's Common Lisp implements `catch` by pushing all the information relevant to the catcher onto the stack to create a *catch frame*. The catch frame encodes the address of the code for the control transfer, the catcher object, and a uniquely tagged catch frame marker which is recognised by GC, the debugger and the throw mechanism. (The RISC implementation actually contains four other fields: one of these holds the constants vector of the function containing the `catch` form, the rest are used to link catch frames together so that `throw` can be implemented quickly). After the code to create the catch frame, the compiler generates code for the body of the catcher as normal, followed by code to pop the catch frame from the stack. Hence, if no `throw` occurs the overhead of a `catch` is just the pushing and popping of the catch frame.

`Throw` is implemented by calling a low-level function to transfer control to the catcher[2], after having first evaluated the result values. This low level function scans the stack looking for the nearest catcher for the specified object. When this catcher has been found, any intervening unwind-protect forms are executed (see section 8.2), and any dynamic bindings are undone (see section 8.3). Finally control is transferred to the address stored in the catch frame.

This implementation relies on a detailed knowledge of the structure of the stack. As with the debugger, I don't know if it is possible to use this mechanism in an AN manner.

8.2 Unwind-protect

It is sometimes useful to guarantee that a piece of code will be executed in a program, even if a transfer of control (such as `throw`[3]) occurs. Common Lisp and Dylan both provide a special form for this called `unwind-protect`. This is commonly used when processing files, for example: a file is normally

1. Dylan has a facility called `bind-exit`, which has similar capabilities. `Bind-exit` is analogous to `catch`, but additionally binds a variable to an exit function. The throw is performed by calling this function, rather than with another language construct.

2. Throw is a Common Lisp special form, not a function, even though it is syntactically very similar to a function. This is because `throw` passes all the values of its second form to the catcher. A normal function call would discard all but the first value.

opened, processed and then closed, but closing the file should always happen even if a throw occurs. Common Lisp actually provides a macro to do this, which expands into code similar to the following:

```
(let ((stream (open "file")))
  (unwind-protect
     (process stream)
    (close stream)))
```

The first form inside the `unwind-protect` (the *protected form*) is executed in an environment in which the remaining forms (the *cleanup forms*) are marked for execution even during a transfer of control. If no such transfer occurs, the `unwind-protect` form returns the value(s) returned by the protected form, after first executing the cleanup forms. One of the cleanup forms may itself transfer control, in which case there are defined semantics about which catchers are still visible. In this case, any other intervening cleanup forms must still be executed. In all cases, the cleanup forms are executed in the context of the `unwind-protect` form, and not in the context of the thrower.

`Unwind-protect` is implemented in terms of a specially modified catcher in Harlequin's RISC Common Lisp[1]. The protected form is executed in the context of a catcher which will catch **any** throw; this is called a *catch all*. The compiler inserts an implicit throw of `nil` to this catcher immediately after the protected form for the case when there is no control transfer. The cleanup forms are executed immediately after the catch all, and hence are always executed. After execution of the cleanup forms, a test is made of the target object for the throw. If this was not the compiler-generated throw then the throw is redone with the original target and value(s), so that a "real" throw can transfer to a point before the `unwind-protect`.

This implementation may also be possible for use with TDF, (once a mechanism has been found for catch and throw), since it makes use of other primitives, and adds little extra complication.

8.3 Special binding

When variables are introduced, or *bound*, in Common Lisp (with the `let` and `let*` special forms), they are normally lexically scoped, as is often the case with static languages such as C. The programmer may instead specify that the variables are to be introduced with *special binding*[2], which gives the variables dynamic scope - i.e. the variables are visible to any function called while the binding is active. The variable values must revert to their previous values, either at the end of a binding form, if a transfer of control terminates a binding form prematurely.

It is possible to implement special binding using either a *shallow binding* or *deep binding* technique (or a mixture of the two). If deep binding is used, a new location is created for the variable each time it is bound, (often this location will be on the stack) and the most recent location must be searched for when the variable is referenced. If shallow binding is used, the same location is always used for the variable (so references are faster), but the old value must be stored somewhere just before the binding (again, usually the stack), and restored just after the binding.

---

3. Other Common Lisp control transfer mechanisms such as `go` and `return-from` are also required to respect the semantics of `unwind-protect`. These mechanisms are lexically scoped, but may have similar dynamic properties to `throw` if used inside a closure. Harlequin's compiler will detect these cases, and use the same mechanism as `throw` for the transfer when necessary.

1. The CISC implementation actually uses a more specific mechanism, but I see no point in describing anything other than the most general solution.

2. Special binding is not provided in Dylan.

It is possible to implement shallow binding with the `unwind-protect` primitive described above, so special binding may not add any complexity to the requirements for TDF. However, special binding starts to become more complex when the notion of *threads* is introduced. Threads are not a standard part of Common Lisp, but most implementations provide them. Where threads are provided, it is intuitive for special bindings to be local to the thread in which they were bound. If deep binding is used, a context switch to another thread does not require any work to adjust the bindings, since the variables are found on the stack, which must be switched anyway. However, if shallow binding is used, the central variable location must be kept up to date each time a context switch occurs, in case a special binding must be done or undone. Normal `unwind-protect` cleanup forms should not be executed at context switch time, however, so another mechanism must be used to implement special binding if threads are provided.

Harlequin uses a shallow binding scheme to implement special binding[1]. The location of a special variable is always in the *symbol value* field of the symbol used to name the variable. When a special binding occurs, the information for each symbol being specially bound is pushed onto the stack, to create a *binding frame*. All binding frames are linked together on the stack for ease of chaining along the frames when context switching. A global location in memory called the *dynamic environment pointer* points to the most recent binding frame.

When a binding frame is created, the contents of the dynamic environment pointer are first pushed onto the stack. Next, for each variable being specially bound, the old value of the variable is pushed onto the stack, followed by the name of the variable (a Lisp symbol); the new value of the variable is then stored in the symbol value field. Finally, the stack pointer is copied into the dynamic environment pointer to complete the link of binding frames.

On exit from the binding form, the bound variables have their old values restored by popping each symbol from the stack, and using this popped value to point to the symbol value field, where the old value is popped. After all the symbols have been popped, the binding frame link is popped into the dynamic environment pointer.

If a control transfer occurs from inside to outside the binding form, the throw mechanism is responsible for undoing the special bindings. All the information to do this is stored on the stack, so the implementation is straightforward. Similarly, during a context switch, special bindings must be undone for the old thread, and redone for the new thread. These two operations are actually performed by the same function, which follows the chain of binding frames on the stack, reverting any bindings in the process. As the function processes each binding frame, it swaps the linking pointers, so that instead of a chain of bindings from newest to oldest, the stack contains a chain from oldest to newest. When the bindings are processed for the new thread, the binding chain starts with the oldest first, so bindings are recreated in the correct order. After processing, the chain again has the most recent binding first.

It is unclear what TDF primitives, if any, are needed for special binding. Extra primitives are only required for a language which supports both special binding and multiple threads, which is not true for any of the standard Lisp language definitions. I hope that TDF will be able to support this type of binding scheme as it may become important with a future language implementation. Hopefully, whatever scheme is adopted to support other operations with dynamic scope (such as `unwind-protect`) will be suitably general that special binding can be implemented too.

1. Although a hybrid shallow and deep binding mechanism is under development for a parallel Lisp.

8.4 Evaluation order

Most Lisp languages (including Common Lisp and Dylan) define the order of evaluation or arguments to function calls and for special forms. Normally the ordering is simply lexical from left to right. Unlike C and many other languages, Lisp compilers are not permitted to optimise this evaluation order unless the compiler can be certain that it will not affect program execution.

This requirement appears to be well supported already in TDF. The `sequence` primitive guarantees order of evaluation.

8.5 Inlining

Common Lisp provides two declaration mechanisms which affect the inlining of function calls: `inline` and `notinline`. `Inline` suggests to the compiler that the programmer would like the function to be inlined. It is free to be ignored. This causes no problems with TDF: the producer can generate TDF for the called function inline without worrying about whether a call is actually performed.

The compiler is **not** free to ignore the `notinline` declaration. *[B92]* points out that there is no explicit way to forbid a TDF producer from inlining a function, but since all named Lisp function calls will be through procedure variables, this is an implicit mechanism for forbidding the inlining, since the variable value can only be resolved at run time. Hence there should be no requirement to extend TDF to forbid inlining.

8.6 Packages

Common Lisp has a module system which allows separate namespaces for symbols. These namespaces are called *packages*. Packages are first class objects in Common Lisp, and are named by strings. Various facilities are provided to manipulate them. The main purpose of packages is to provide a modularized mapping between symbols (which are objects) and their ASCII printed represented representation, as seen and produced by the Lisp reader and writer respectively.

Each Common Lisp symbol has a package associated with it (its *home* package); this may be `nil` if the symbol has no package, in which case it is not possible to find the symbol from its name by using the reader.

Each package may export symbols so that they may be accessed from other packages; such symbols are said to be the *external* symbols of the package, while other visible symbols are *internal*. Package “A” may *use* package “B”, so that all the external symbols of “B” are visible inside “A”. A *shadowing* mechanism is provided so that a package can import all the external symbols of another package apart from the explicitly shadowed ones. Symbols from other packages may also be made visible to a package on a symbol by symbol basis by explicitly importing them. A symbol need only be visible from a package to be exported; it need not have that package as its home.

Harlequin’s Lisp implements packages as composite structure objects containing the package name, a hash table of internal symbols, a hash table of external symbols and a list of used packages. This implementation is effectively defined in portable Common Lisp, and should require no extra TDF features. However there are some interesting bootstrap issues, since packages are required to resolve the names of symbols stored in FASL files, and hence that the code required to create the first packages cannot be loaded from a FASL file itself. Bootstrapping is described in more detail in Section10 below.

9. Program Errors

9.1 Conditions

The Common Lisp Condition System gives the language an object oriented exception handling system. *Conditions* are a means of describing exceptions. They are defined in terms of objects which inherit from the class `condition`. Some common condition classes are `warning` and `error`, but programmers may define their own classes.

A condition is raised by passing a condition object to the function `signal`[1]. A facility is provided to set up *handlers* for conditions. `Signal` will invoke the most recently established handler which matches (i.e. is a superclass of) the condition object. After examining the condition object, this handler may cause a control transfer to some code which is designed to handle the error; alternatively, the handler may just return, in which case the next handler is found. If no handler transfers control, `signal` just returns. The function `error` is defined to call `signal`, and then to enter the debugger if nothing handles the condition.

In addition to condition handlers, it is possible to set up *restarts*. Each restart is a function which provides some means of continuing from an exception. Restarts may be invoked by handlers, or they may be invoked interactively by the debugger. Functions are provided to find all the restarts which have been established, and to find if there is a restart with a given name.

Harlequin's condition system is implemented using more primitive control transfer mechanisms (ultimately catch and throw), and special binding. It's implementation does require the addition of a conditional throw mechanism, which works like `throw` if a catcher exists, but returns `nil` if there is no catcher. Implementation of the condition system in TDF should cause no special problems, since it is entirely built on these lower level primitives.

9.2 Stack bounds check

The design of the language of Common Lisp is ideally suited for developing new software. The interpreter, interactive debugger and the dynamic updating facilities make it simple to experiment. This is a very different way of working from a C language programmer, who would normally have to edit, recompile, relink and restart a whole program after the results of a debug session. One consequence of this way of working is that Lisp systems must be error tolerant. If the Lisp environment crashes because of a programmer's bug, then all the benefits of dynamic updating are lost.

A common problem with untested software is stack overflow. There is no stated requirement that a Common Lisp system should be able to recover from this, although recovery is standard in commercially available Lisps.

Harlequin's Lisp compiler inserts code for a stack check on entry to each function[2]. This does not necessarily increase the code size, since the stack check can be performed out of line by one of the dedicated parameter processing functions. If a stack overflow is detected, an error is signalled, and restarts are provided to increase the size of the stack in case the programmer wishes to continue with the computation.

1. This is a slight simplification as `signal` can create the condition object itself if it is passed more primitive arguments.

2. It is possible to disable this check, by setting the `safety` level sufficiently low with a declaration. Of necessity, parts of the error system itself must do this.

There are other advantages of this stack checking. The entry to a function is known to be a safe time to interrupt Lisp; there are guaranteed to be no spurious values around to confuse GC. Hence, by suitably modifying the stack limit variable, a Lisp computation will be forced to interrupt itself and call the error handler. If the error handler is sufficiently smart it can detect whether there was a real stack overflow, or whether there was an interruption for another reason. This mechanism is used by Harlequin Lisp to handle interrupts from the operating system timer so that threads only context switch at GC safe times. Similarly it is used to enter the debugger at a GC safe time when a keyboard break (e.g. Control C) occurs.

This stack checking technique is important to the robustness of the Lisp implementation as a programming environment. However, there would appear to be no way to specify this behaviour in an AN manner. It might be possible to use these techniques if tokens are provided in TDF for setting and reading the stack limit and stack pointer, but this would mean that the TDF must confirm the existence of a stack in its AN model. The restart mechanism for growing the stack is not a vital part of stack checking, but is useful nevertheless. Presumably this would require other TDF extensions to create a new stack - although this could possibly be handled by the GC code, since addresses on the stack would have to be fixed up anyway.

9.3 Traps

Another common error caused by an incorrect program is division by zero. This normally causes a processor trap which is handled in an operating system defined manner. Other processor traps might include memory access violations in a virtual memory environment; these are possible if a programmer has compiled erroneous code using a low safety declaration.

A good Lisp implementation will handle these errors. This is done by using an operating system specific mechanism to initialize a trap handler, which eventually calls the Lisp `error` function.

Since the initialization of the trap handler is inherently operating system specific, this problem is really outside of the domain of TDF.

10. Common Language Extensions

10.1 Multiple threads

We have already seen some of the implementation details of Harlequin's threads extensions to Common Lisp, and some of the stack manipulation facilities required. Part of the implementation of a threads extension concerns the operating system interface (e.g. to request timer interrupts), which is therefore out of the scope of TDF.

One further point is worth mentioning. Since threads may be dynamically created and destroyed, storage for the threads should be handled by the garbage collector. This storage may include some data structure to describe the status of the thread, and it will also include a stack for the thread. Hence stacks become first class objects in their own right, and their presence can no longer be "hidden under the carpet".

10.2 Saving images

Since Lisp programs are created by dynamically loading code and data into a smaller Lisp program, it is normal to be able to save the current state of the program (or *image*). The language definition for Common Lisp does not describe how this should be done, but normally a function is provided which does a full garbage collection, before saving the current state as an executable program to a file. An alternative approach

would be to save the heap data of the image only[1], so that a small, static executable program can reload it. This technique has the advantage that it is less operating system specific, since it does not require any knowledge of the object file format.

At first sight, it would seem that it should be possible to specify saving the entire heap area in an AN manner. However, this will presumably require an explicit TDF token, since TDF abstracts away the idea of an area of memory called a heap. The TDF requirements for saving images are probably shared with the requirements of the garbage collector, which also must consider memory as an addressable area.

11. Bootstrapping

In section 7.1 we saw that Lisp programs are built up from top level forms, which have side effects on the Lisp environment, such as setting the value of a symbol, or associating a symbol with a function. Normally these top level forms are compiled as anonymous functions requiring no arguments, and are executed when a FASL file is loaded.

The FASL loader code can obviously not be loaded by the FASL loader itself, so how is it initialized? We have seen that the FASL loader is user extensible, and so it must directly interact with the Lisp language model. It would seem to make sense to implement the FASL loader in Lisp (although this is not a requirement). If the Lisp compiler is used to generate this bootstrap code, it must generate output in pure TDF form for direct manipulation by the TDF linker and installer. It must be possible to specify initialization code in a TDF stream, in order to create Lisp compatible functions, and call them during the bootstrap. Presumably a token will be required to express this in TDF. Such a token would be required for other languages with local file or module initialization too.

12. Optimisations

12.1 Stack allocation

The design of Lisp type languages imply that any implementation must provide a storage management mechanism (allocator and GC). There are defined functions in the language for allocating objects, but unlike C or Pascal, there is no way of explicit deallocating. In general, this is a good idea since calculating when an object is no longer required is a complex problem and has been the source of many bugs in many programs. In Lisp, when an object is allocated, it is assumed that it will have an indefinite lifetime; the garbage collector guarantees that objects will be freed when they are no longer needed. With modern GC design the penalty for this can be low.

Of course, there are times when it is easy to know an object can be deallocated, and deallocating it as soon as possible reduces memory requirements and possibly GC time. The most common example of this is when an object has dynamic extent. For example, the object is created on entry to a function, it is processed, and then the object is no longer required at the end of the function. This situation is very common in most programming languages and most programs. Sometimes, the compiler can detect that the lifetime of an object has dynamic extent[2]. In the general case, only the programmer can know. To optimise this

1. Note that most Lisp code is likely to be in the heap area.

2. For example, the list of arguments created for the &rest parameter will have dynamic extent unless the list is passed to a function which might assign the list to a data structure with a more permanent lifetime.

common case, Common Lisp defines a `dynamic-extent` declaration which the programmer may use to inform the compiler that the object can be deallocated explicitly if the optimization is available.

Harlequin's Lisp optimizes allocation which is declared to have dynamic extent by allocating the object on the stack rather than on the heap. All objects on the stack inherently have dynamic extent, so this is a low cost mechanism. The implementation requires the compiler to generate stack allocation code, which must respect any tagging conventions for type checking and GC. The garbage collector must also know how to handle objects which are encountered on the stack since they cannot be relocated. In addition, compiled code for functions which stack allocate must be able to correctly pop the stack afterwards. Since the stack allocated objects can include variably sized arrays and lists, this is normally a matter of saving and restoring the stack pointer. But if the stack is itself a garbage collectable object which can move, the stack pointer must be saved as a relative offset from the frame pointer, since the garbage collector will preserve the integrity of the frame pointer. Note that a frame pointer is always used for functions which stack allocate, as indeed it is used for all but leaf-case functions. This means that variables on the stack can always be addressed even after allocation on the stack.

Stack allocation makes a very dramatic difference to the performance of most Lisp programs. A Lisp implementation is free to ignore the `dynamic-extent` declaration, but if optimizations are never performed for dynamic allocation then the implementation will never be worthy of serious use. Note that this is not an optimisation that can be detected automatically in the installer - the producer must indicate when dynamic allocation is possible. Again, coding this in TDF may be difficult since there are no defined stack operations. If stack allocation is impossible, it may still be useful to heap allocate the objects but to have a specific deallocation function, called implicitly from compiled code. This would not require any TDF extensions, but would lose some of the benefits of stack allocation, as deallocation would be slower.

13. Portability Issues

13.1 Constants at compile time

TDF can potentially make Lisp compiler output portable in a way which is impossible using dedicated compilers for each architecture. As with other languages, there is no way that TDF can guarantee that all programs will be portable, and some interesting portability problems arise in Lisp.

Mostly these problems are also encountered when cross-compiling to a different target architecture, which is an area fairly well understood by Harlequin. Compiler technology can help to reduce these problems by maintaining a separate environment for the target architecture - called the *remote environment*. In the case of compilation to TDF, the remote environment should contain AN descriptions of constants and an AN object representation, whereas the local environment will hold AS descriptions, resolved by the TDF installer itself at load time.

The design of Common Lisp makes it impossible to ensure that the correct environment will always be used. For example, consider Common Lisp macro functions, which take the original form as an argument, and return a modified form as a result, performing an arbitrary computation in-between. The remote environment may have a specific copy of a macro function which is designed to produce code for the target environment, but unfortunately, the macro function itself must run in the host environment. This may introduce portability problems if the non-portable results of the macro function are inserted into the target code. Harlequin has developed a set of guidelines for programmers who wish to avoid these problems.

There some further problems which may be encountered when the remote machine is architecture neutral. Consider the compilation of a function which contains as a constant a large integer. The integer is of such a size that on some platforms it would appear as an immediate quantity (i.e. a fixnum), but on other platforms it must be represented as a pointer (i.e. a bignum). If we assume a function representation where constants are stored in a constants vector, it seems reasonable to assume that the TDF installer will be able to determine if a fixnum representation is suitable or not, and insert either the integer or a bignum pointer into the constants vector. However, on some architectures it may be appropriate to represent a fixnum directly in the code stream as an instruction operand. In this case, the integer should be omitted from the constants vector, and the location of all constants located later in the vector should be shuffled along by 1. It may be possible to code this operation in TDF, but it sounds as though the coding is very complex.

14. Conclusions

We have seen that the dynamic properties of Lisp require implementation techniques which are not used for compiling static languages like C or Pascal. The interaction with the program and the garbage collector requires that function frames can be found easily on the stack. The requirement for run time checking of arguments passed to a function suggests a different calling convention from C, and the requirement for proper tail recursion optimization probably rules out the use of the C convention. In addition, Lisp has the ability to dynamically compile and load code, and for code and constants to be independently relocatable.

Support for these features will require extensions to TDF, either in the form of new primitives, or by defining new tokens. It is hoped that the information in this document will help with deciding exactly what TDF extensions are required.

15. References

[B 92] David Bruce, “Architecture Neutral Compilation of Common Lisp”, COOTS working paper, DRA Malvern, April 1992.

[Dylan] Apple Computer, “Dylan an object oriented dynamic language”, April 1992.

[S 90] Guy L. Steele Jr., “Common Lisp The Language, second edition”, Digital Press, 1990.

[TDFspec] DRA Malvern, “TDF Specification”, September 1992